DRAL
Daresbury Laboratory
Rutherford Appleton Laboratory

RAL Report
RAL-94-118

Client Server Architecture for MultiDatabase Access Ingres SQL Server

M Dixon

November 1994

Rutherford Appleton Laboratory Chilton DIDCOT Oxfordshire OX11 0QX

Client Server Architecture for MultiDatabase Access

Ingres SQL Server

M Dixon

(Visitor to Data Engineering Group
System Engineering Division
DRAL)

Faculty of Information & Engineering Systems
Leeds Metropolitan University
Beckett Park
LEEDS LS6 3QS

Issue Date: 10 October 1994

ABSTRACT

The interoperating of multiple database systems requires a client application which acts as a mediating agent for database server processes; these server processes must be accessed using network procedure calls. This report addresses the problems and associated design options that arise in constructing a dynamic SQL server process for the Ingres Relational Database Management System. The Ingres SQL server has been constructed in C on a Unix platform and accessed across a SunOS network. It is intended that the narrative in this report together with the source code for both the SQL Server and the primitive Client would act as a starting template for a writer of application code that required dynamic SQL access to an Ingres database. The choice of a network protocol is discussed.

Contents

1 INTRODUCTION 1
1.1 Research Context 1
1.2 Heterogeneity 2
1.3 Purpose of Technical Report 3
1.4 Choice of Development Tools 3

2 INTERPROCESS COMMUNICATION 4
2.1 Introduction 4
2.2 External Data Representation 4
2.3 Data Types 5
2.4 Communication Protocol 6
2.5 Client Server Recognition 6

3 INGRES SERVER APPLICATION 7
3.1 Introduction 7
3.2 Registering a Server Application 7
3.3 Processing messages from the Client 8
3.4 Returning messages from the Client 9
3.5 Dynamic SQL 9
3.5.1 Initialising a descriptor block 10
3.5.2 Connecting to the database 10
3.5.3 Connecting to the dynamic descriptor block 11
3.5.4 Non Retrieval Operations 12
3.5.5 Retrieving field definitions 12
3.5.6 Retrieving data values 14

4 CLIENT APPLICATION 16
4.1 Role of the Client Application 16
4.2 Connection to Server 16
4.3 Construction of SQL string for passing to server 17
4.4 Handling of returned data 17

5 CONCLUSION 19

REFERENCES 20

ACRONYMS 21

Acknowledgements: 21

APPENDIX 1 22
A1.1 Compiler / library switches 22
A1.2 Ingres environment variables 23

Client Server Architecture for MultiDatabase Access Ingres SQL Server

1 INTRODUCTION

1.1 Research Context

The development of interoperating database systems has become a major business requirement in recent years. Operational databases tend to be developed to meet specific departmental needs; the use of that data with data from other sources for management information tends to arise after the database has been in place for some time. Also the merger of companies with different databases leads to a need for their information systems to cooperate. Although some system design methodologies such as Information Engineering [Mar89] seek to take a strategic approach to the development of information systems it is unlikely that a fully integrated database management system could be developed for a company that is dynamically adapting to business requirements.Of especial interest to established companies is the migration and integration of mission critical legacy systems [MBrod92, MBrod93].

An extension of the sharing of information systems to allow intercompany data exchange is underway with the use of EDI for ordering and invoicing.

There is also an increasing need to be able to access publicly available database sources such as library catalogues, census data, and travel information. For these databases the information supplier defines the format of their database and it is the responsibility of the accessor to conform to the access protocol.

Following Date's 12 rules for distributed databases [Date90] there has been an underlying assumption that interoperation of information systems should preserve autonomy of the individual information systems. This autonomy should include:

1) design autonomy for data models and schema definitions,

2) communication autonomy where an information system can decide whether to respond to a request from another system,

3) execution autonomy where each information system decides the order in which external and local transactions are performed.

A catagory distinction is made between distributed databases and interoperating databases. Distributed databases are seen as tightly coupled global information sharing systems where the global system has control over local data and processing; the division into subsystems is constrained solely by engineering issues.[CoulOrl93, BriHurPak92]. It is the unitary nature of distributed database management systems that precludes them from tackling the information management issues for which interoperation is required especially since each of the local systems is of the same type. Gateways provide a way in which one dbms can access another dbms however there are limits to the extent to which they can be used since they are dependent on database vendors being willing to incorporate the gateway for a specific dbms on a specific platform; Ingres / Star provides a good example of a distributed database which supports a very limited set of gateways.[IngresV6].

M W Bright, A R Hurson and S H Pakzad [BriHurPak92, see also HurBriPak93] have reviewed current issues in multidatabase systems. They have produced a taxonomy in which they classify Interoperable systems as loosely coupled information sharing systems which are limited to simple data exchange. Their paper summarises by class the wide range of projects on multidatabase systems; identifying whether they are commercial, prototype or research products. Most global data models are reported as relational.

1.2 Heterogeneity

The types of heterogeneity that can occur in databases has been reviewd by Sheth and Larson [ShethLar90]. They identify four types which they catagorise as Hardware, Operating System, Communication, and Database System; the latter has subtypes of Semantic Heterogeneity and Differences in DBMS. A substantial discussion of the these issues is given in that review. Sheth and Larson then developed the three level ANSI/SPARC schema architecture for simple databases into a five level schema architecture for federated databases. The five levels are

1) local schema for a component dbms in its native data model
2) component schema derived by translating the local schema into a common data model for the federated database system
3) export schema identifying what data is being made available to the federation
4) federated schema which combine multiple export schema
5) external schema which defines a user's view of the federation

The appropriate transformation, filter, and constructor processes are reported by them.

Coulomb and Orlowska [CoulOrl93] have studied the implications of semantic heterogeneity for the design autonomy of interoperating information systems and they conclude that generally it is not possible to resolve semantic heterogeneity without violating design autonomy. This leads them to question the utility of research on general architectures.

1.3 Purpose of Technical Report

Embedded sql provides a mechanism whereby a user's application programme can issue sql commands to a database and process the resultant tables; the application programme only requires knowledge of sql. Unfortunately this facility only works for hard coded queries. If an application user wishes to dynamically define a query then dynamic sql must be used to access the database and specific knowledge of the database dynamic data areas is required. Fortunately, for Ingres at least, a model for a terminal monitor application has been provided as part of a tutorial. [IngresC90]. The work undertaken so far has entailed the construction of server task which receives a sql command from a client application and returns the result set to the client.

This report has been prepared to facilitate the construction of a dynamic sql server for the Ingres relational database system. The aim is to provide a template for the writers of interoperating systems so that they have a functioning starting point for the development of their own applications. Associated libraries of source code will be provided.

1.4 Choice of Development Tools

It was decided to begin this work using a relational DBMS since many of the issues related to semantic heterogeneity are manifested in relation database systems. It is also the case that a recent survey of cooperating information systems showed that the global data model most frequently selected in practice for interoperation was the relational model. [HurBriPak93]. The choice of C/C++ and Unix for the server environmnet was determined by the availability at DRAL of SED's Ingres relational database on the commercially widely used Unix platform; C was the preferred programming language of the group reflecting its wide commercial use. The creation of the server was greatly facilitated by the availability of tutorial code from Ingres for a terminal monitor application which could be used as a starting template. [IngresC90]. Initial compatibility lead to the choice of Unix and C for the client application as well.

The choice of the relational approach is particularly appropriate for network intercommunication because of its set nature. One SQL command retrieves a set of results for one network call. This can significantly reduce the amount of message passing compared to navigational data models.[Date83].

2 INTERPROCESS COMMUNICATION

2.1 Introduction

This section discusses the way that data is represented as it is passed between applications, the choice of a communications protocol, and how the server waits for a message from the client.

2.2 External Data Representation

In order to transfer data between computers the client and the server need to agree the format of that data. XDR is a standard for the machine independent description and encoding of data.[NPG37, NPG103]. XDR can be used for the communication of data between such different machines as Sun Workstation, VAX, IBM-PC, and Cray.

There is a Sun xdr sub-library of primitive functions for handling basic C data types. Access is made available through

```
#include <rpc/rpc.h>
```

at the beginning of a programme.

For a basic C data type xxx the format of the xdr function call would be

```
XDR *xdrs;
xxx   *xp;                                          /* example int *ip  */
if (! xdr_xxx (xdrs, xp) ) { return ( FALSE); } /* example xdr_int(xdrs, ip) */
                            return( TRUE);
```

xdrs is the XDR stream handle and is never inspected or modified.
xp is a pointer to a variable of type xxx.
xdr_xxx is of type bool_t which has been declared as type integer as follows

```
#define bool_t int
#define TRUE 1
#define FALSE        0
```

A list of the primitives can be found in [NPG110] and a formal definition is available in [NFXDR]. An attractive feature of these xdr functions is that the same function can be used for conversion in the client and in the server; the direction of conversion being set by the context.

When an xdr routine has to be supplied even though no data is to be passed then the function

```
bool_t xdr_void();                    /* always TRUE */
```

is used.

A complex function enables the passing of strings.

```
u_int maxlength;          /* maxlength is the maximum length of the string */
char **sp                 /* a pointer to the pointer for the string */
bool_t        xdr_string(xdrs, sp, maxlength);
```

Another complex function enables the passing of a vector.

```
xxx  * vectorp;           /* pointer to start of vector of type xxx */
int numelements;          /* number of elements in vector           */
bool_t        xdr_vector( xdrs, vectorp, numelements, sizeof(xxx), xdr_xxx);
```

The specific function that issues the call to the server, clnt_call(), permits one xdr function with the associated address for the data to be sent; clnt_call() also has one xdr function with the associated address for the data to be received in reply from the server. A programmer can construct their own xdr functions to act as arguments for clnt_call(). Essentially such a function consists of an ordered set of calls to the xdr library functions. The manipulation of the combinations xdr functions follows the usual rules for a C function; this enables the programmer to pass within their constructed xdr function data which is used to process following data in the same transfer. In the server application setting the pointer to the data to NULL will allow the xdr primitive to allocate the required memory dynamically.

2.3 Data Types

There is a range of data types supported for the server database. Since update to the database occurs through a SQL command string constructed dynamically in the client and dispatched by the server to the dbms there is no inherent problem in using the full range of the database's datatypes. However each database type needs to be mapped to a local type system for reporting results. Such a mapping should ensure that there is no data loss and that the subsequent processing is as simple as possible. The mapping used here follows that illustrated by Ingres; it trades simplicity for extra storage.

Database Type	C Language Type
integer	long int
float	double
money	double
char	string
varchar	string
date	string[25]

The consequence is that the client application wo ıave a relatively simple task in merging the types extracted from different server dbmss.

2.4 Communication Protocol

The RPC (remote procedure call) protocol is independent of the transport protocols; it deals only with the specification and interpretation of the messages and not how a message is passed. Sun RPC is currently supported on both UDP/IP and TCP/IP transports. An extensive discussion of the relevant protocols has been given by Black.[Black94]. The criteria for selection of one transport rather than another are given in [NPG36,NPG37]. TCP/IP was choosen for the following reasons

1. The size of the results could exceed 8 Kbytes

2. Updates to the database must be performed only once otherwise different data values could be generated in the database.

3. The application needs to maintain a high degree of reliability.

Use of TCP/IP means that the application does not have to implement its own retransmission policy and that a reply from the client means that the remote procedure was executed once and only once; if no reply is received then the client cannot assume that the procedure has not been executed. Even with TCP/IP it is necessary to handle server crashes. The server will keep track of each open client connection so server machine resources will limit the number of simultaneous clients for TCP/IP.

2.5 Client Server Recognition

Client application recognition of a Server application is based upon the concept of a port which is a logical communication channel in a host. The server application waits on the port until it receives messages from the network. A server application registers with its host computer using the port mapper service to allocate a port. A client gets the server application's port from the server's host portmapper. The client application can then call the server application with a message that contains the port number [NPG21,NPG22].

3 INGRES SERVER APPLICATION

3.1 Introduction

The server application is based upon the Terminal Monitor Application which is fully listed in Ingres documentation. [IngresC90]. It is not intended to document that code here; we are concerned with the aspects that relate to enabling a user to access the column definitions and data values in order to pass them across the network to a client application.

3.2 Registering a Server Application

It is first necessary to create a TCP/IP service transport handle for sending and receiving messages; the transport is associated with a socket. svctcp_create() creates a stream socket providing bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. [NPG288]Using RPC_ANYSOCK causes a new socket to be created and the socket is bound to a local TCP port.[NFRPC_SVC_CREATE(3N)]. TCP based calls use buffered i/o so users may specifiy the size of the buffers with sendsz and recvsz; values of 0 choose defaults.

```
SVCXPRT  *transp;                    /* transp is a pointer to a data structure */
u_int sendsz;                        /* size of send buffer */
u_int recvsz;                        /* size of recive buffer */
transp = svctcp_create(RPC_ANYSOCK, sendsz, recvsz); /* NULL if fails */
```

The next step is to erase the portmapping of this server application programme with its version number; this is to deregister any possible remnants from a previous run of the server application which may have crashed.

```
u_long prognum = PROGNUMBER;
                              /* The program number - also known to client  */
                              /* eg 0x20000008 [NPG37] */
u_long versnum = VERSIONNUM;
                          /* The version of PROGNUMBER -known to client */

pmap_unset(prognum, versnum);
```

It is now appropriate to register with the local portmap service the server application identified by prognumber and versionnum. The application is associated with a dispatch

procedure, void faccess(); this is the procedure that is executed when the server application receives a meesage. The application is registered with the TCP/IP protocol by setting the value of protocol. [NFRPC_SVC_CALLS(3N)]

```
u_long protocol = IPPROTO_TCP;
                        /* The interprocess protocol to be used is TCP */
void faccess();/* void function but void *faccess()  in documents */
svc_register( transp, prognum, versnum, faccess, protocol );
                        /* returns TRUE if succeeds else FALSE */
```

The dispatch procedure has the following form

```
void faccess(rqstp, transp)
struc svc_req *rqstp;
SVCXPRT    *transp;
{
 see below for main body of this function which accepts an
 arguement and sends a reply to the client
}
```

3.3 Processing messages from the Client

The final part of the main function of the server application is the function void svc_run() which waits for incoming messages to arrive and invokes the dispatch procedure faccess(). The function svc_run only returns in the case of some errors. [NFRPC_SVC_REG(3N)]

The message arrives on the stream pointed to by transp and is extracted in function faccess() using the function

```
bool_t svc_getargs( transp, xdr_xdr0, strp)

SVCXPRT *transp;
int  xdr_xdr0;  /* the routine used to decode the incoming args */
char ** strp;                   /* The pointer to the  address pointer for where the
            arguments will be placed*/
char * pstr[2]; /* pointers to the strings comprising the message; here it is assumed that the
message consists of two strings with the first string being the operational code identifying the
message type and the second string being the detailed message.    strp = &pstr;  */
```

The string pointed to by pstr[0] can be used in a test condition to see which dynamic sql feature is being invoked [connect | sql | disconnect].
The string pointed to by pstr[1] can be used as [database name| sqlcommand | null].

3.4 Returning messages from the Client

Messages are returned to the client on stream transp using function

bool_t svc_sendreply(transp, xdr_proc, strp)
[NFRPC_SVC_REG(3N)]

SVCXPRT *transp;
int xdr_proc; /* the routine used to encode the outgoing data */
char ** strp; /* The pointer to the address pointer for where the arguments will be placed*/
char * pstr[2]; /* pointers to the strings comprising the message; here it is assumed that the message consists of two strings with the first string being the operational code identifying the message type and the second string being the detailed message. strp = &pstr; */

For a simple return with an error code/ completion condition then the use of strp can mirror its use in function svc_getargs(). However, where there is a significant amount of processing and the returned data is complicated then xdr_proc can be quite complicated. Although forming a matched pair with the svc_getargs() function the behaviour is somewhat different if the data to be transfered exceeds the buffer sizes. It would appear that there is an asynchronous breakout from xdr_proc and a bufferload of data is sent to the client before processing resumes within xdr_proc. This is of significance in handling data of unknown quantity from a database. A consequence may be that the matching decoding xdr routine in the client handles the data rather than waiting for it to be passed to the calling routine. Under these circumstances strp becomes a dummy.

3.5 Dynamic SQL

Embedded sql provides a way of executing sql statements from within a host 3gl programming language, in our case within C/C++; this assumes that the sql command is hard coded and compiled. Dynamic SQL allows an application to construct a SQL command while running, store the SQL command within a host string variable, and execute the SQL command. A handbook of the ISO standard for SQL has been produced by Cannan and Otten [CanOtt93]. SQL statements fall into two categories; those that do not retrieve result data from the data base (such as delete, insert, create, drop) and those that process data results returning from the data base (such as select and fetch).[IngresC 90, p4-1 et seq].

Source files which need to be preprocessed by the Ingres C utility ESQLC are given the extension sc. The compilation and linking must occur in an environment in which the Ingres paths are set. [See Appendix 1,2]. This is normally done by the developer creating a

command tool window and remotely logging into the workstation on which Ingres is run; the environment varaibles are then set.

SQL commands can be embedded almost anywhere in a program where host commands are allowed [OSQLR3-21]. Commands begun by the phrase "EXEC SQL" are embedded commands.

3.5.1 Initialising a descriptor block

The first embedded SQL command must be

```
EXEC SQL INCLUDE SQLCA;
```

This command incorporates SQL's error and status handling mechanisms, known as the SQL Communications Area (SQLCA) into the program. This area is used for control purposes by such commands as WHENEVER [OSQLR5-72]

```
EXEC SQL WHENEVER SQLERROR STOP;
```

3.5.2 Connecting to the database

Connection of the server to a database eg mdtest, is via the command

```
EXEC SQL CONNECT mdtest;
```

For connection to a dynamically defined database name the DECLARE SECTION command must be used for the database name.

```
# define        DBNAME_MAX              50      /* max size of database name */
EXEC SQL BEGIN DECLARE SECTION;
char dbname[DBNAME_MAX + 1];
EXEC SQL END    DECLARE SECTION;

fgets(dbname, DBNAME_MAX, stdin);       /* collect from keyboard */

EXEC SQL CONNECT :dbname;                               /* connect to database */
```

The server may sever connection to the database by disconnecting via the command

```
EXEC SQL DISCONNECT;
```

3.5.3 Connecting to the dynamic descriptor block

A host language descriptor area, called the SQLDA is a structure that has been defined by Ingres. SQLDA holds descriptive information about the fields required for the sql command.

In order to include the SQLDA descriptor block in an application the code should contain the statement

EXEC SQL INCLUDE SQLDA;

The SQL statement that has been dynamically constructed needs to be placed in an especially prepared character array which is made known to SQL by the DECLARE SECTION statment. A command name for and a cursor for the command name are also made known through the commands DECLARE STATEMENT and DECLARE CURSOR respectively. In the example below stmt and csr should not be regarded as host language variables. The DECLARE STATEMENT declares a name used in the program to identify prepared SQL statments. It is regarded as a declarative command. The DECLARE CURSOR FOR names a cursor for use with a specified statement name. It is a compile time command that must appear before the first line that refers to the cursor.

The command name is dynamically connected with the contents of stmt_buf through the prepare command. The return data definitions are then placed in SQLDA using the DESCRIBE command; it retrieves length and name information about a prepared select command into the SQLDA. [OSQLR5-28]. It is necessary to allocate memory to receive these returned data definitions; this is done by setting up a default sized area and then extending if the actual need exceeds that default.

```
EXEC SQL DECLARE stmt STATEMENT; /* dynamic sql command name */
EXEC SQL DECLARE csr CURSOR FOR stmt; /* cursor for stmt */
```

```
# define STMT_MAX  1000  /* max SQL statement size */
EXEC SQL BEGIN DECLARE SECTION;
char stmt_buf[STMT_MAX + 1]; /*statement dynamically constructed in this*/
EXEC SQL END    DECLARE SECTION;

# define DEF_ELEMS  5                    /* default number of return columns */
Init_Sqlda( DEF_ELEMS)    /* create memory to store column defs */
EXEC SQL PREPARE stmt FOR :stmt_buf;  /* Prepare the command name */
EXEC SQL DESCRIBE stmt INTO :sqlda;
                /* If the number of returned columns is greater than space reserved
```

```
then free space,  and reserve appropriate amount of space
                    sqlda->sqln is number of return columns for which space exists
sqlda->sqld is number of columns for which space is wanted */
if(sqlda->sqld  > sqlda->sqln)
                { Init_Sqlda(sqlda->sqld);
                  EXEC SQL DESCRIBE stmt INTO :sqlda;}
```

Function Init_Sqlda (num_elems) frees and reserves memory

```
void Init_Sqlda(num_elems)
int             num_elems;                  /* number of return columns */
{
if(sqlda) free( (char *)sqlda); /* Free old SQLDA */
                                            /* Allocate new SQLDA */

 sqlda = (IISQLDA *) calloc(1,
 ( IISQDA_HEAD_SIZE + (num_elems * IISQDA_VAR_SIZE)  );
sqlda->sqln = num_elems;
}
```

3.5.4 Non Retrieval Operations

Non SELECT SQL commands do not return data rows to the user so it is not necessary to handle the return data field definitions. Under these circumstances the SQL command can be executed directly with only a status code being returned to be detected by WHENEVER SQLERROR and the number of rows affected.

```
int rows;                                   /* number of rows affected */
if ( sqlda->sqld == 0 )                     /* test for no return rows      */
{
EXEC SQL EXECUTE stmt;
rows = sqlca.sqlerrd[2];
```

3.5.5 Retrieving field definitions

The number of result columns is then straightforwardly obtained from

```
long int numcolumns;                    /* number of result columns */
numcolumns = sqlda->sqld;  /* sqld is a two byte integer. */
```

The details of the column are held in an array, sqlvar[]. It is first necessary to set up a pointer of type IISQLVAR to the descriptor area;

```
IISQLVAR * sqv;
char * p_namestr;
```

For column i we obtain the value of the pointer as follows

```
sqv = &sqlda->sqlvar[i];
```

The column name and the length of the array containing the name can then be obtained as follows

```
colnamelength = sqv->sqlname.sqlnamel;
p_namestr = &sqv->sqlname.sqlnamec;
sqv->sqlname.sqlnamec[colnamelength] = '\0';  /* convert to string */
```

This is different from the manual page 4-9 which does not split sqlname into the name and length parts. The name is a character sequence and is converted to a string by adding a trailing null. The column name length should not exceed 34 characters although Ingres allows names of <=24 characters. Unfortunately the name does not maintain track of the table so an implied order has to be used to handle that information where it is significant; ie two columns in different tables with the same name, eg date, but semantically different would not be distinguished in the naming.

Each column is associated with a data type and a numeric code is used to indicate which data type.

Type	Code	Code variable
integer	30	IISQ_INT_TYPE
float	31	IISQ_FLT_TYPE
char	20	IISQ_CHA_TYPE
varchar	21	IISQ_VCH_TYPE
date	3	IISQ_DTE_TYPE
money	5	IISQ_MNY_TYPE

A negative value for the code indicates that the data values could be null.

```
int base_type;
base_type = sqv->sqltype;     /* the value of the data type code */
if (base_type < 0 ) base_type = -base_type;
```

A switch on the base_type then allows the identification of the correct storage area for that column type.

```
long int columnwidth;                    /* storage required for data type */

switch (base_type)
        { case IISQ_INT_TYPE:
                columnwidth = sizeof( long);
          break;
                /* the length of an array is stored in the structure
                   pointed at by sqv->sqllen; */
         case IISQ_CHA_TYPE:
         case IISQ_VCH_TYPE:
                columnwidth = sqv->sqllen + 1;
          break;
          case IISQ_MNY_TYPE:
                columnwidth = sizeof( double);
          break;
          case IISQ_FLT_TYPE:
                columnwidth = sizeof( double);
          break;
          case IISQ_DTE_TYPE:
                columnwidth = DATE_SIZE + 1;
                                                /* define DATE_SIZE 25 */
          break;
        }
```

3.5.6 Retrieving data values

Once the SQLDA area has been DESCRIBED and the return columns defined it is possible to retrieve the data values.
The SQL SELECT statement is initiated when the cursor, csr, is opened using the command [OSQLR5-49]

```
EXEC SQL OPEN csr;
```

Each row can now be retrieved using the command

```
EXEC SQL FETCH csr USING DESCRIPTOR :sqlda;
```

A FETCH is the only way to position the cursor on a row [OSQLR5-39]; a cursor must be closed before reopenning it.

Each row is then retrieved through the use of a fetch command. The data value for column i+1 is stored in the array sqlda->sqlvar[i]. The number of rows returned is not knowable in advance so a count needs to be kept. Fetching can continue while the sqlca.sqlcode ==0. The row retrieval loop is covered by failure condition SQLERROR which allows for failure handling; in this case no more data leads us to close the cursor.

```
long int intval;/* returned integer value */
double realval;                 /* returned real value */
char * p_strval;                /* pointer to returned string */
EXEC SQL WHENEVER SQLERROR GOTO Close_Csr;
numrows = 0; /* Initialise counter on number of rows retrieved */
EXEC SQL OPEN csr;          /* Opens the cursor for this select statement */
while ( sqlca.sqlcode ==0 )
 {EXEC SQL FETCH csr USING DESCRIPTOR :sqlda;
                                /* The row is fetched into the descriptor area */
                    numrows ++;/* increment retrieved row counter */
 for ( i=0; i < numcolumns; i++)                /* loop over returned columns */
                   { sqv = &sqlda->sqlvar[i];
                     if( (base_type = sqv->sqltype) < 0 ) base_type = - base_type;
switch (base_type)
                                {
                   case IISQ_INT_TYPE:
                                intval = *( long *)sqv->sqldata;
                    break;
                                /*  p_strval type (char *) is pointer to string */
                    case IISQ_DTE_TYPE:
                    case IISQ_CHA_TYPE:
                    case IISQ_VCH_TYPE:
                                p_strval = sqv->sqldata;
                    break;
                     case IISQ_MNY_TYPE:
                                realval = *(double *)sqv->sqldata;
                     break;
                     case IISQ_FLT_TYPE:
                                realval = *(double *)sqv->sqldata;
                     break;
                                } /* case */
                   } /* columns */
} /* rows */

Close_Csr:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL CLOSE csr;
```

4 CLIENT APPLICATION

4.1 Role of the Client Application

The role of the client application is to collect SQL commands from the user and to dispatch them to the appropriate server application as a string. In order to do this it needs to identify the host name of the computer on which the server application is running as well as the specific programme number and version of the server application. For reasons of reliability discussed in Section 2 above a TCP/IP stream socket connection is established between the client and the server. The SQL command is passed to the server using xdr and the reply containing the data is received using xdr. At the end of a session the client's handle is destroyed and the socket is closed.

4.2 Connection to Server

Our template for the connection to the server host and application is adapted from that given in [NPG75] for use of UDP/IP. In our case we wish to use TCP/IP so that the client is linked to the server application using a stream socket; this provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. (In the Unix domain stream sockets are nearly identical to pipes [NPG280]).

The pointer to host computer is obtained by use use of a library function call:

```
struct hostent *hp;             /* hostent is defined on p291 Network Programming*/
hp = gethostbyname("apple"); /* apple is workstation running Ingres */
```

The client handle is created [NFRPC_CLNT_CREATE(3N)]

```
int sock = RPC_ANYSOCK /*The socket is system issued through RPC */
bcopy(hp->h_haddr, (caddr_t)&server_addr.sin_addr, hp->h_length);
server_addr.sin_family = AF_INET;  /* socket is in Internet domain */
server_addr.sin_port = 0;                    /* remote portmapper to find address */

register CLIENT *client;                     /* a handle for the client      */
u_long      prognum;                         /* program number on host */
u_long      versnum;                         /* version number on host */
u_int       sendsz;                          /* size of send buffer          */
u_int       recsz;                           /* size of receive buffer       */
            /* create a client handle; beware this function
            succeeds if host has an earlier! version of the programme */
```

```
client = clnttcp_create(        &server_addr, prognum, versnum,
                                &sock, sendsz, recsz);
```

A call may then be made to the server application passing a message
[NFRPC_CLNT_CALLS(3N)]

```
u_long procnum = PROC_NUMBER; /* Can be used as case switch to select a
                procedure associated with client handle */
struct timeval total_timeout;
char ** strp;                   /* pointer to outgoing strings */
int xdr_xdr0();/* to serialise outgoing data  - user defined */
char ** retp;                   /* pointer to results          */ .
int xdr_xdrResult();            /* to deserialize the returned result */
total_timeout.tv_sec = 60; /* time in secs allowed for response from server */
total_timeout.tv_usec= 0;
clnt_call(client, procnum, xdr_xdr0, strp, xdr_xdrResult, retp, total_timeout);
```

Different xdrResult functions are used for different expected return arguments.

At the end of the client session the handle can be destroyed, associated private data areas deallocated and the socket closed.

```
clnt_destroy(client);
```

4.3 Construction of SQL string for passing to server

The approach adopted in this prototype assumed that the client would send two strings to the server each time. The first string would identify the type of message and the second would contain the substantive SQL .

Message type	Message Content	Function
Connect to Database	Database name	Get_Db_Name(strp, pstr)
SQL command	Command string	Get_SQL_Stmt(strp,pstr)
Disconnect from Database	Treated as SQL command	"

4.4 Handling of returned data

The set of fields returned is determined dynamically by the query so each query had a series of field names, field types, and field lengths returned. The designer has considerable choice in deciding what is the best way of handling the returned data. In this prototype it was

decided to choose the simplest available approach. The data from the database is of unknown size unless a count of records was made before the data itself was retrieved. It was decided that a double pass over the data was not acceptable so a convention was used that associated a rownumber with each row from the database. A conventional negative value for the row number was used to indicate no more rows. Adopting this approach meant that the size of the receiving arrays in xdrResult in the client application had to be created by dynamic allocation rather than using the inbuilt facilities of the xdr primitives.

Example of how the data was transfered

```
Inside xdrResult(xdrsp, retp)
XDR             *xdrsp;                         /* handle for transport */
char            **retp;                         /* pointer to return data */
{
long int      numcolumns;      /* number of columns retreived */
long int  *p_numcolumns;     /* pointer to numcolumns */
p_numcolumns = &numcolumns;
xdr_long(xdrsp, p_numcolumns);              /* deserialize  number of columns */

int maxsize = 36;                           /* maximum size of column name */
char ** p_columnname;                       /* pointer to column name strings */
long int * p_columnwidth;    /* pointer to column width vector  */




/* allocate space for for arrays */
p_columnname = (char **) malloc ( numcolumns * maxsize * sizeof(char));
p_columnwidth = (long *) malloc ( numcolumns * sizeof(long));

int i;

for (i= 0; i < numcolumns; i ++)
                {
xdr_string(xdrsp, &p_columnname[i], maxsize);              /* deserialize col name */
xdr_vector(xdrsp, p_columnwidth, numcolumns, sizeof(long), xdr_long);
                                                           /* deserialize col widths */
                }
retp[0] = (char *) p_numcolumns;
retp[1] = (char *) p_columnname;
retp[3] = (long int *) p_columnwidth;
}
```

Recall that xdrReseult() is an argument to clnt_call() with (char **) retp as the return data pointer.

```
#define MAX_RET_COLS  50                /* maximum number of data columns */
char **retp;                            /* return data pointer to pointers */
char  *retstring[MAX_RET_COLS + 6];     /* pointers to header, data */
long int numcolumns;
char ** ph_columnname;                  /* pointer to column name string */
long int * ph_columnwidth;   /* pointer to columnwidth  */

retp = &retstring[0];
numcolums = *(long *) retstring[0];          /* number of columns returned */
ph_columnnname = (char **)retstring[1]; /* pointer to column name string */
ph_columnwidth = (long int *) retstring[3]; /* pointer to columnwidth vector */
```

5 CONCLUSION

This paper has described the way that a client server architecture has been produced for the Ingres relational database on a Unix platform using the C programming language. The network protocol TCP/IP was chosen for interprocess communication.

REFERENCES

Black94 U Black TCP/IP and Related Protocols, McGraw Hill, 1994, ISBN 0-07-005553-X

BriHurPak92 M W Bright, A R Hurson, and S H Pakzad, A Taxonomy and Current Issues in Multidatabase Systems, Computer Vol 25 (3) p50, 1992

Brod92 M Brodie, The Promise of Distributed Computing and the Challenges of Legacy Systems, BNCOD10, p1, 1992

Brod93 M Brodie Interoperable Information Systems: Motivations, Status, Challenges and Approaches, VLDB, Tutorial, 1993

CanOtt93 S Cannan and Gerard Otten, SQL - The Standard Handbook for ISO 9075, McGraw Hill, 1993, ISBN 0-07-707664-8

CoulOrl93 R E Coulomb and M E Orlowska,Interoperability in Information Systems, Technical Report 263, Department of Computer Science, University of Queensland, 1993

Date90 C J Date, Introduction to Database Systems Vol 1,p621, 1990, Addison Wesley, ISBN 0-201-52878-9

Date83 C J Date, Introduction to Database Systems Vol 2,p303, 1983, Addison Wesley, ISBN 0-201-14474-3

HurBriPak93 AR Hurson, M W Bright, & S H Pakzad Multi database Systems : An Advanced Solution for Global Information Sharing IEE Computer Society Press 1993, ISBN 0-8186-4422-2

IngresC90 Ingres, Embedded sql Language Companian Guide for C January 1990, E-4

IngresV6 Introduction to Ingres V6.4 Chap 2, p16, 1991

Mar89 James Martin, Information Engineering Book 1 Introduction, 1989, Prentice Hall

NPGxx Network Programming Guide, Sun MicroSystems, 1990, pxx

NFXDRxx Network Functions, XDR(3N), Sun MicroSystems, 1990, pxx

NFRPCxx Network Functions, RPC(3N), Sun MicroSystems, 1990, pxx

OSQLRx-y Open SQL Reference Manual, Sun MicroSystems, 1989, px-y

ShethLar90 A P Sheth & J A Larson ACM Surveys, Vol 22 (3), 183, 1990
Federated Database Systems for Managing Distributed,
Heterogeneous, and Autonomous Databases

ACRONYMS

ANSI/SPARC	American National Standards Institute / Systems Planning and Requirements Committee
BNCOD	British National Conference on Databases
DBMS	database management system
DRAL	Daresbury Rutherford Appleton Laboratory
EDI	electronic data interchange
FIES	Faculty of Information and Engineering Systems
ISO	International Standards Organisation
LMU	Leeds Metropolitan University
RPC	remote procedure call
SED	Systems Engineering Division
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol/Internet Protocol
xdr	eXternal data representation
UDP/IP	User Datagram Protocol/Internet Protocol
VLDB	Very Large Database Conference

Acknowledgements:

It is a pleasure to acknowledge the help given by colleagues in the Informatics Department at DRAL during this study. In particular John Kalmus and Keith Jeffery for guiding the overall direction of work; Kevin Lewis for much help and assistance with Ingres; Ines Day for helping with systems problems. Zie Jhang of LMU FIES provided advice on remote procedure calls.

APPENDIX 1

A1.1 Compiler / library switches

The following shows the precompilation of the source code with the esqlc preprocessor for the C language for a file containing the main application.

```
esqlc tmasvr4.sc            /* precompile file tmasvr4.sc and dependent files */
cc -c tmasvr4.c             /* apply C compiler to generated files */
ld -dc -dp -e start -X -o tmasvr /usr/lib/crt0.o tmasvr4.o \
$II_SYSTEM/ingres/libingres.a \
-lm -lc                     /* link modules accessing functions in ingrs library
                               II_SYSTEM is an environment variable
                               for /home/ingres                              */
```

A1.2 Ingres environment variables

The following describes the environment which was used to compile and link the code.

```
HOME=/home/nfs8/1/md
SHELL=/bin/csh
TERM=sun
USER=md
PATH=/home/ingres/ingres/bin:
/home/ingres/ingres/lib:
/usr/lib:/usr/openwin/bin:
/usr/openwin/bin/xview:
/usr/openwin/demo:
/home/ingres/ingres/bin:
/home/ingres/ingres/lib:
/usr/lib:
/usr/ucb:
/bin:
/usr/bin:
/etc:
/usr/etc:
/usr/ral/bin:
/home/nfs8/1/md/bin:.
LOGNAME=md
PWD=/tmp_mnt/home/nfs8/1/md/Termon
OPENWINHOME=/u/OpenWin3
II_SYSTEM=/home/ingres
KEYBD=sun4-E
TERM_INGRES=wview
ING_EDIT=/usr/openwin/xview/textedit
PRINTER=all
DISPLAY=pecan:0.0
TERMCAP=vs|xterm|vs100|xterm terminal emulator (X window system):li#24:co#80:  :
:cr=^M:do=^J:nl=^J:bl=^G:le=^H:ho=\E[H:
:co#80:li#65:cl=\E[H\E[2J:bs:am:cm=\E[%i%d;%dH:nd=\E[C:up=\E[A:
:ce=\E[K:cd=\E[J:so=\E[7m:se=\E[m:us=\E[4m:ue=\E[m:
:md=\E[1m:mr=\E[7m:me=\E[m:      :ku=\EOA:kd=\EOB:kr=\EOC:kl=\EOD:kb=^H:
:k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:ta=^I:pt:sf=\n:sr=\EM:
:al=\E[L:dl=\E[M:ic=\E[@:dc=\E[P:        :MT:ks=\E[?1h\E=:ke=\E[?1l\E>:
:is=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;6l:
:rs=\E[r\E<\E[m\E[2J\E[H\E[?7h\E[?1;3;4;6l:xn:
:AL=\E[%dL:DL=\E[%dM:IC=\E[%d@:DC=\E[%dP:
:ti=\E7\E[?47h:te=\E[2J\E[?47l\E8:
:hs:ts=\E[?E\E[?%i%dT:fs=\E[?F:es:ds=\E[?E:
WINDOWID=13631510
```