# *The Implementation of Cooperative Mechanisms among System Components in a Heterogeneous Multidatabase Environment*

Jiansan Chen, Omran A. Bukhres, and
Ahmed K. Elmagarmid
Purdue University

ABSTRACT: The cooperative interaction of component database systems and auxiliary system components is a prerequisite to the smooth functioning of a heterogeneous multidatabase system. The institution of mechanisms to foster such cooperative interaction is therefore a significant factor in the development of multidatabase systems. Issues to be considered in this regard include establishing and managing communication channels and information flow among system components, synchronizing their executions, and detecting and recovering from errors. This paper presents a practical solution of these issues, illustrated with detailed examples of its implementation in UNIX environments. Examples drawn upon experience gained in the development of the InterBase System [7], an ongoing heterogeneous database project at Purdue University. The implementation of the proposed solution in non-UNIX environments, such as IBM CMS and MS-DOS, is also illus-

trated. Fundamental aspects of InterBase, including logical architecture and component interoperability, are described briefly.

## 1. Introduction

The computing environment found in most contemporary organizations is characterized by distributed and heterogeneous software systems. The distribution of these systems reflects the diverse nature of modern business, while their heterogeneities arise in the process of fulfilling diverse computational and information processing requirements. Originally, these systems ran in isolation to support their individual applications. It soon became evident that more complex applications involving multiple systems could be supported if inter-systems cooperation could be fostered. While the integrity of the pre-existing systems must be preserved, there is at the same time an increasing demand for information sharing on an organization-wide or region-wide basis, thus further increasing the need for effective system integration.

Heterogeneous databases are a prime example of global applications requiring the cooperation of component software systems. A heterogeneous database is designed to support global applications accessing more than one component database or other software system. Unfortunately, these systems were not originally designed to facilitate such cooperation and there is as yet no general model for interoperability among such isolated software systems. There is therefore a pressing need for the development of an environment to support global applications involving multiple software systems.

The issue of cooperation among database systems has been extensively studied and many prototype systems have been developed. Examples include ADDS [5], BellCore Project [2], Carnot [26], DATAPLEX [9], DOM [6], DQS [3], EDDS [4], Linda [27], Mermaid [25], MRDSM [19], Multibase [18], NDMS [14], Odu [21], OMNIBASE [23], Pegasus [1], Proteus [12], Scoop [13], Sirius-Delta [11], SWIFT [16], Unibase [15], and VIP-MDBS [17]. Other approaches, such as

Ingres Gateways [22] and Sybase Open Servers [24], have also been offered as solutions to the problem of cooperation among database systems. Most of these methods, however, focus on high-level issues and contain few of the practical details of low-level implementation essential to their acceptance in industrial settings.

In this paper we present a detailed implementation of the InterBase System [7], a practical approach to the problem of cooperative interaction among multiple software systems running on different machines. The purpose of InterBase is to provide global information sharing and to facilitate access to local systems in a distributed and heterogeneous environment. By allowing users to write global applications over component local systems without modification, InterBase carries the potential for great increases in productivity and improvement in global applications processing. The following proved to be critical to the implementation of such a distributed system:

- **Coordination among system components.** In a distributed system, an application is typically carried out by several system components running on different machines, these components must synchronize their execution to operate effectively. While some components must run continuously, others must coincide with the life-cycles of an application. Such a dynamic system necessitates a finely-tuned communication protocol. Furthermore, as the underlying computer platforms may provide inadequate support for coordination and synchronization among system components, effective mechanisms must be developed to accomplish these goals within the existing context. This issue will form the main thrust of this paper.
- **System portability.** As any cooperative system will be most useful if it is executable on a variety of computer system platforms, portability is an important issue. InterBase is currently implemented primarily on UNIX platforms, with some components implemented on IBM mainframes and PCs. This implementation has been designed to use only those UNIX interfaces supported by all UNIX platforms, such as UNIX V5, SUN/OS, AIX, HP-UX, and OSF/1. InterBase can therefore be executed on different UNIX platforms with at most minor modifications, as well as on MACH platforms, as MACH is a superset of UNIX. To preserve flexibility, only UNIX network

facilities, such as TCP sockets, have been used to implement the communication protocol among InterBase components running on different machines, and the more restrictive high-level functions such as SUN RPC (Remote Procedure Calls) facilities have been avoided. The use of TCP sockets provides several benefits. First, they provide reliable communication among participating processes, ensuring that all messages will be sent to their destination, an issue critical to all distributed applications. TCP sockets also comply with UNIX file I/O operations, allowing programmers to conveniently process network messages of different sizes. TCP sockets are supported with a uniform format by all UNIX platforms, providing portability across platforms. Furthermore, they are supported by the TCP/IP network communication protocol [10], an industrial standard provided or simulated by many computer platforms such as UNIX, IBM VM/CMS, PC-DOS, VMS, and Macintosh System 7. Finally, they support a client/server communication model which is suitable for communication among dynamic and static system components. In this model, during a communication process, continuously running components can act as servers, while other components can take the role of clients.

- **Experience gained in the process of implementation.** A number of significant insights were reached which should prove to be useful in the design of further systems; all these issues will be shared in this paper.

Though the current implementation of InterBase is in C, other programming languages may also be used to implement components of InterBase, as long as the object code is compatible with that generated by the C compiler and/or provides interfaces to network facilities using TCP/IP.

The body of this paper is organized as follows. Section 2 provides an overview of the InterBase System and discusses cooperative mechanisms among InterBase components. The implementation of these cooperative mechanisms in UNIX environments is set forth in Section 3. Implementation issues in non-UNIX environments are discussed in Section 4. Finally, Section 5 presents conclusions gleaned from our current investigations and outlines directions for future work.

## 2. *InterBase Components and Their Coordination*

The InterBase System is a heterogeneous multidatabase prototype system developed at Purdue University, which has been designed to provide an execution environment for global applications over distributed, heterogeneous, and autonomous software systems. This system has been widely demonstrated in the last three years and has been recently enhanced with additional features, such as a decentralized concurrency controller. There are two major components to the InterBase System, as depicted in Figure 1: the Distributed Flex Transaction Manager (DFTM), and a set of Remote System Interfaces (RSIs). The DFTM is at the center of InterBase. It consists of a set of DFTM replicas, each responsible for the consistent and reliable execution of a global transaction over the entire system. The DFTM provides a distributed task specification language, the InterBase Parallel Language (IPL) [8] with which users can specify a global transaction. RSIs are specially designed InterBase agents which are superimposed on the individual Local Software Systems (LSSs). RSIs provide a uniform system-level interface between the DFTM and LSSs, buffering the heterogeneity of the LSSs and thus relieving the DFTM from dealing directly with each LSS. A user can invoke a high-level interface, currently under development, to write a query to InterBase; the interface will translate the query into an IPL text, and the text will then be sent to the DFTM for execution. A user with a good grasp of the LSSs and a fluency in IPL can also write and send IPL texts to the DFTM for direct execution. A graphical interface, InterBaseView [20], will be also provided over InterBase to aid users in writing and executing global transactions in IPL. The RSI Directory stores information such as location and allowable data transfer methods for different RSIs, and thus supports location and distribution transparency for the system.

Interactions among InterBase modules are presented in Figure 1, where arrows indicate the data and control flow. Currently, InterBase runs on an interconnected network with a variety of hosts, such as UNIX workstations and IBM mainframes, and supports global applications accessing many local software systems including SAS, Sybase, Ingres, DBS, and UNIX utilities. The InterBase System represented in Figure 1 is a simplified logical architecture; in practice, it can be tai-

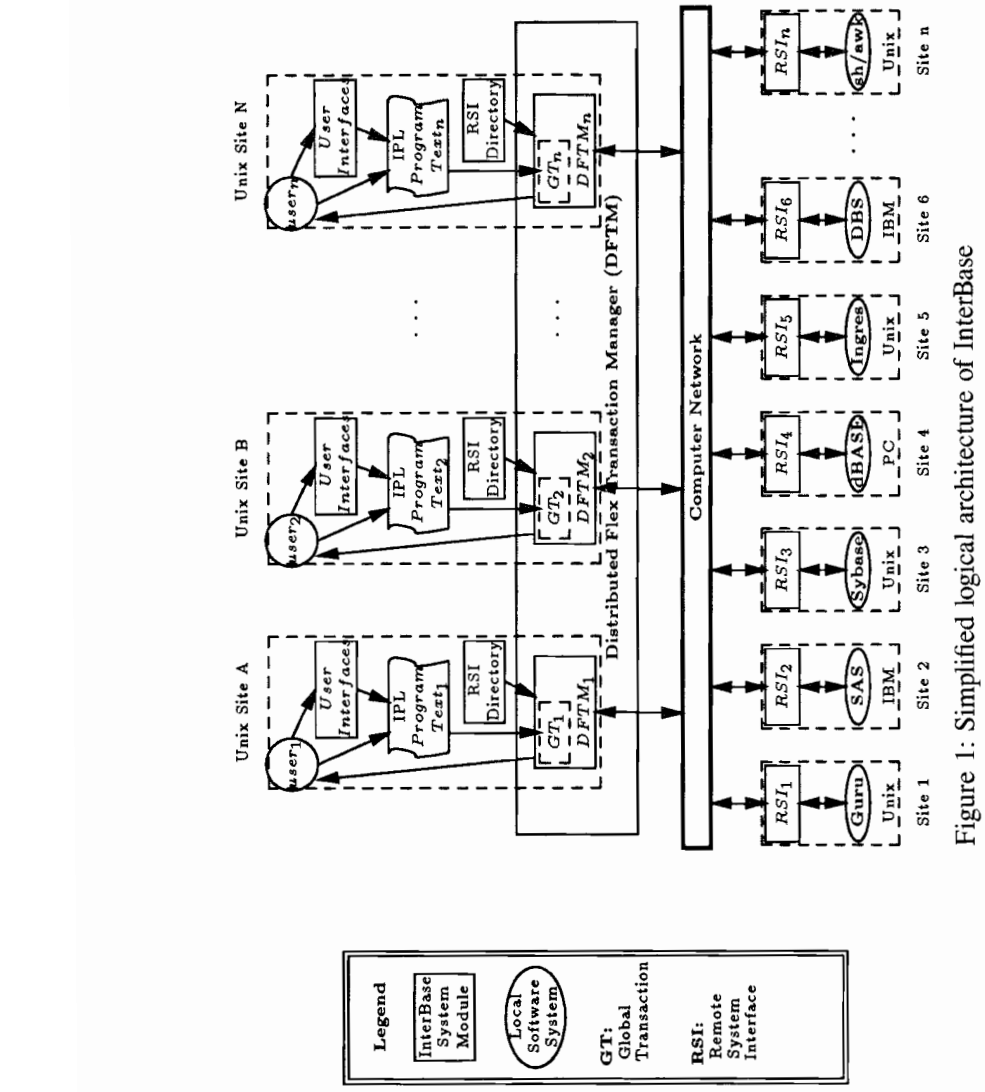


Figure 1: Simplified logical architecture of InterBase

lored to meet different physical environments. For example, if there are several Ingres database systems incorporated into InterBase, a distinct RSI must be provided for each. More than one RSI can be run simultaneously on a given machine.

The general strength of this architecture is its decentralized nature, which can be seen in Figure 1. The DFTM is distributed over all the machines from which IPL programs are executed; that is, each global transaction is associated with a DFTM replica, which is responsible for the consistent and reliable execution of the global transaction. For example, IPL program $Text_1$ is executed as the global transaction $GT_1$, which is in turn carried out by the DFTM replica $DFTM_1$, as illustrated in Figure 1. Exchange of information within Interbase is performed via computer network, and thus each module of InterBase has location transparency. InterBase is designed to avoid direct communication among DFTM replicas; each DFTM replica can therefore be executed independently. The implementation of DFTM replicas is thus greatly simplified. (Throughout this paper, DFTM replicas and global transactions will be used as interchangeable terms.)

Each global transaction consists of subtransactions, each of which must be executed on an LSS through its associated RSI. As the first step in its execution, a global transaction $G_i$ must therefore communicate with the relevant RSIs to arrange the relative execution order of its subtransactions on the appropriate LSSs. The relevant RSIs then execute the subtransactions of $G_i$ in the specified order.

An RSI consists of an RSI server and RSI services. The RSI server accepts the execution requests of concurrent global transactions and interacts with the transactions to arrange for the execution order of their subtransactions on its associated LSS. It then creates RSI services for these subtransactions according to the specified order. In this way, InterBase allows several DFTM replicas to be executed concurrently as long as their execution is serializable, thus increasing the throughput of InterBase. An RSI service is responsible for the consistent and reliable execution of its associated subtransaction and is coincident with the life-cycle of that subtransaction. The RSI server also monitors the status of running and completed RSI services, as an aid in scheduling the execution of upcoming and queued subtransactions.

RSI servers are LSS-independent, in that they do not actually make contact with their associated LSSs. RSI servers for different

LSSs can therefore share the same codes and are independent from their associated RSI services. On the other hand, RSI services are LSS-specific. That is, each type of RSI service must be designed specifically to work with its associated LSS. This versatility is an advantage of dividing the RSI function between RSI servers and RSI services. A second benefit of this division is that it permits concurrent execution of subtransactions in InterBase. Furthermore, all RSI servers are capable of running on UNIX platforms, regardless of the location of their associated RSI services. This type of RSI structure facilitates both communication among DFTM replicas and RSI servers and crash detection of InterBase components.

The activity of InterBase components is dynamic, in that it adapts to suit the nature of the currently executing global transactions. As an illustration, let us envision an instance in which there are three global transactions $T_1$, $T_2$, and $T_m$ running in InterBase. $T_1$ consists of a subtransaction on Sybase database 1 and the UNIX shell of file system 2. $T_2$ consists of a subtransaction on Sybase database 1, Ingres database n, and the UNIX shell of file system 2. $T_m$ consists only of a subtransaction on Ingres database n. Assume that their execution order in each site has been determined as follows: (1) in Site 1, $T_1 \rightarrow T_2$; (2) in Site 2, $T_1$ and $T_2$ can run concurrently; (3) in Site n, $T_2$ and $T_m$ can run concurrently. The dynamic state of DFTM replicas, RSI servers, and RSI services at a given time is illustrated in Figure 2.

We see that global transaction $T_1$, executed by the DFTM replica $DFTM_1$, communicates with RSI $server_1$, RSI $server_2$, RSI $service_{1,1}$, and RSI $service_{1,2}$. $T_1$ communicates with RSI $server_1$ and RSI $server_2$ to determine its execution order on Sites 1 and 2 and to request its execution. RSI $service_{1,1}$ and RSI $service_{1,2}$ are created by RSI $server_1$ and RSI $server_2$, respectively, and communicate directly with $T_1$ to execute the corresponding subtransactions. The execution of $T_1$ therefore requires the coordination of $DFTM_1$, RSI $server_1$, RSI $server_2$, RSI $service_{1,1}$, RSI $service_{1,2}$, Sybase DMBS, and the UNIX shell, most of which are independent system processes running on disparate sites. Note that Site 3 contains only RSI $server_3$, since there is as yet no subtransaction on Site 3. Note as well that, because data stored in Database 3 and Database n may not be identical, they have been provided with different RSIs, although they both are Ingres sites and their RSIs may be identical in their codes.

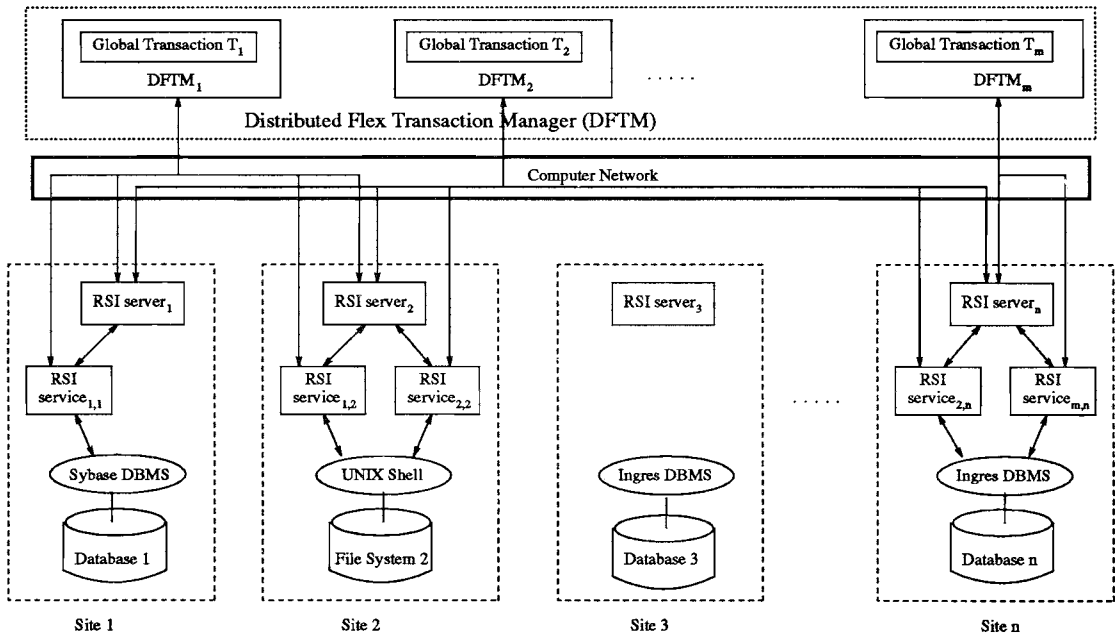


Figure 2: Typical dynamic state of DFTM replicas, RSI servers, and RSI services

From this example, it is evident seen that RSI servers are static components of InterBase, while DFTM replicas and RSI services are both dynamic components. While RSI servers are required to run continuously, a DFTM replica is coincident with the life-cycle of a global transaction, and an RSI service is coincident with the life-cycle of a subtransaction.

Figure 1 illustrates the distribution of the copies of the RSI Directory among all InterBase machines on which DFTM replicas may run. While such distribution complicates directory updating, it does lead to improved performance. Updates are performed only on the infrequent occasions when an LSS is added to or removed from InterBase.

## 3. *Implementing Cooperative Mechanisms among InterBase Components in UNIX Environments*

To support the InterBase architecture and interactions among InterBase components described in Section 2, efficient mechanisms for InterBase component coordination must be developed. In this section, we shall provide details regarding the implementation of such mechanisms in UNIX environments.

### *3.1 Generic "interbase" User and "interbase" User Group*

All InterBase components are managed on behalf of a generic interbase user with the login name `interbase`. That is, all InterBase files in InterBase machines must be under the control of the user `interbase`. All InterBase users are assinged to user group[1] `interbase`, and the group ID of InterBase files is set to `interbase`. These files should be executable only by their owner `interbase` and by InterBase users. The set-user-ID privilege for these files should be activated, so that the permission modes of InterBase executable files appear thus:

```
-rws--x--  1 interbase interbase
  53896 Oct 23 14:56 interbase ...
```

1. User groups are defined in file /etc/group.

Once an InterBase user invokes a DFTM replica for the execution of an IPL program, that user effectively becomes the generic user `interbase`. All DFTM replicas are thus run on the same effective user ID, which is `interbase`, thus simplifying both the access control and the coordination among InterBase components. Since the number of users in a group is indefinitely expandable, InterBase can be accessed by any user who joins the `interbase` user group.

### *3.2 Establishing Communication Channels between a DFTM Replica and RSIs*

As indicated in Section 2, a DFTM replica is responsible for the execution of a global transaction, while an RSI is responsible for the execution of subtransactions sent by DFTM replicas to its associated LSS. DFTM replicas and RSIs must therefore coordinate to execute global transactions. Because DFTM replicas and RSIs can be executed on different machines, communication between them must be ensured. Two factors enter into this process. First, RSI servers are static components of InterBase, while DFTM replicas and RSI services are dynamic components. Furthermore, a given RSI server processes execution requests from several DFTM replicas. The client/server model is therefore best suited to specify the communication channel between DFTM replicas and RSIs. In communications following this model, each RSI server acts as a server, while a DFTM replica acts as a client. The following gives a detailed approach to establishing such communication channels.

Each RSI server is assigned a TCP port in the file "`/etc/services`" on the machine at which the RSI server is executed. This TCP port becomes the medium with which DFTM replicas communicate with the RSI server. For example, if an Ingres RSI server named `ingres1` must be executed on machine `ector`, the following or a similar statement must be included in the file "`/etc/services`" on machine ector:

```
ingres1 2200/tcp
```

To avoid confusion, TCP port 2200 must not be assigned to any other server, and there is no other server named `ingres1` in file "`/etc/services`". That is, TCP port 2200 must be assigned only to RSI server `ingres1`.

After being started, an RSI server obtains a socket for its TCP port defined in the file "/etc/services" using C program routine get_listen_socket[2]. The RSI server can execute an infinite loop, allowing it to accept messages sent to the socket:

```
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

...

get_listen_socket() /* binding a socket to specified
  TCP port, in server mode */
{
    struct servent *sp;      /* service entry */
    struct sockaddr_in sa;   /* socket address */
    struct hostent *phe;     /* host entry */
    u_short port;
    int     fdListen;
    char localhost[MAXHOSTNAME+1];

    /* assume RSI_name specify the name of the RSI server */
    if ((sp = getservbyname(RSI_name, "tcp")) == NULL)
        Error_Handler("getservbyname");

    gethostname(localhost, MAXHOSTNAME);
    if((phe = gethostbyname(localhost)) == NULL)
        Error_Handler("gethostbyname");

    sa.sin_family = phe->h_addrtype;    /* network type */
    sa.sin_port = sp->s_port;            /* get the port */
    sa.sin_addr.s_addr = INADDR_ANY;
      /* accept connection from all */

    if ((fd = socket(phe->h_addrtype, SOCK_STREAM, 0)) < 0)
      /*create a TCP socket*/
        Error_Handler("socket");
```

2. The C program routines and segments used in this paper are simplified to emphasize specific points; the actual programs are much more complicated.

```
    /* assigns a name to the socket */
    if (bind(fdListen, &sa, sizeof (struct sockaddr_in)) < 0)
        Error_Handler("bind");

    if (listen(fdListen, SOMAXCONN) < 0)
      /* set up for listening */
        Error_Handler("listen");

    return (fdListen);
}
```

The TCP port assigned to each RSI server must also be known to the DFTM replicas, because this port presents the dedicated communication medium between these two components. This may be accomplished by defining the same port in the file "/etc/services" for the RSI server on the machines where DFTM replicas are generated. A DFTM replica can thereby obtain the TCP port for the corresponding RSI server by invoking a system call getservbyname and can connect to the socket of the relevant RSI servers, by using C program routine connect_to_RSIs:

```
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

 ...

extern struct subtransaction  *subtransQueue;
  /* the head of subtransaction queue */

connect_to_RSIs() /* the routine for a DFTM
  replica to connect to RSI servers */
{
    struct subtransaction *sup;
    struct sockaddr_in sa;       /* socket address */
    struct hostent *phe;         /* host entry */
    struct servent *pse;         /* service entry */

    /* for all subtransactions */
    for (sup = subtransQueue; sup != NULL,
     sup = sup->t_nextSubtrans) {
```

```
            /* obtaining the TCP port for the
              specified RSI server */
            if ((pse = getservbyname(sup->rsiName,
              "tcp")) == NULL)
                Error_Handler("getservbyname");
                  /* error handling */
            else
                sup->tcpPort = ntohs (pse->s_port);

            if ((phe = gethostbyname(sup->hostName))
              == NULL)
                Error_Handler("gethostbyname");

            sa.sin_family = phe->h_addrtype;
              /* network type */
            sa.sin_addr = *( (struct in_addr *)
              (phe->h_addr));
            sa.sin_port = htons(sup->tcpPort);
              /* get the port */

            if ((sup->tcpFd = socket (phe->h_addrtype,
              SOCK_STREAM, 0)) < 0)
                Error_Handler("socket");
                 /* error handling */

            /* connect to corresponding RSI server */
            if (connect(sup->tcpFd, &sa, sizeof(struct
              sockaddr_in)) < 0) {
                perror("connect");
                activate_RSI(sup, &sa);
                  /* activate the RSI server */
            }
        }
    }
```

The DFTM replica can then use file I/O operations to send/receive information to/from relevant RSI servers. Although communication is established between DFTM replicas and RSI servers, the generation of an RSI service by a relevant RSI server using a `fork` system call delegates all the resources of the RSI server to the RSI service. The communication channel set up between the DFTM replica and the RSI server is thus inherited by the RSI service. This UNIX property greatly simplifies the implementation, although potential confusion must be avoided by the following careful coding:

```
if (fork() != 0) { /* parent process, still
  RSI server */
    close(fdConn);
      /* close the communication channel between the
          DFTM replica and granted subtransaction */
    ...
}
else {  /* child process, RSI service */
    close(fdListen);    /* close the TCP port */
    ...
}

...
```

That is, RSI server must close its communication channel to a DFTM replica, after generating an RSI service for that DFTM replica. The RSI service must in turn close the socket for the TCP port.

### *3.3 RSI Server Activation*

If an RSI server can be activated as a standard server of its underlying computer system when the system starts, that RSI server is *statically activated*. The static activation of an RSI server can be specified with the appropriate shell statement in file "`/etc/rc`"; the file is executed automatically when the computer system is booted. However, because the static activation of RSI servers cannot always be guaranteed, we have provided a mechanism to allow the DFTM to *dynamically activate* an RSI server whenever necessary. After activation, RSI servers are left continuously operating. The dynamic activation of RSI servers also provides support for RSI server crash recovery. In UNIX, dynamic activation is easily implemented, as illustrated in the following description.

InterBase maintains an RSI directory containing the execution files of the DFTM. The RSI directory holds communication parameters of various RSIs, including their locations, the proper communication protocols, allowable data transfer methods, invocation formats, and timeout parameters. For example, information regarding locations may take the form of an Internet hostname. Timeout parameters are used to handle timing problems and can usually be obtained from application statistics. The RSI directory therefore provides a systematic approach to the creation of location and invocation format transparency of RSIs

among different networks. The following is a sample entry from the RSI directory for an Ingres RSI on UNIX. The first column of each item is the keyword for the item:

```
ingres1@ector  {  /* the RSI name used as key */
  PROTOCOL: tcp, ftp;  /* support both tcp and ftp
    protocols */
  HOSTNAME: ector.cs.purdue.edu;
    /* communication channel */
  TCPPORT : 2200;
    /* the tcp port used for communication */
  USERID  : "interbase";  /* remote RSI user id
    and encrypted password */
  PASSWD  : "KHGCHLEAEIDLMKGEJK";
  CMD     : "/usr/interbase/rsi_ingres/start" $PASSWD;
    /* invocation path */
  TIMEOUT : 60 seconds;
    /* maximum response time for the RSI */
  RETRY   : 3;       /* the number of retries
    for timeout before giving up */
}
```

The following is the shell script file

```
/usr/interbase/rsi_ingres/start
```

for the activation of the Ingres RSI:

```
#! /bin/sh
# activate an Ingres RSI
# change to the correct directory
DIR='/bin/echo $0 | sed -e s/start/./'
# if the directory exists
if [ -d $DIR ];
then
    cd $DIR
# send the directory message back
    echo $DIR
# if the executable file for the RSI exists
    if [ -f ingres.rsi ];
    then
# execute the file with the database ingres1
  and the given password
        nohup ingres.rsi ingres1 $1 >
          LOGFILE 2>&1 &
```

```
# send the execution message back
        echo nohup ingres.rsi ingres1 $1 ">"
          LOGFILE "2>&1 &"
        exit
    fi
fi
# send the failure message back
echo no ingres RSI for database ingres1
```

With the help of the RSI directory, an RSI server can be easily activated remotely by a DFTM replica, using C program routine `activate_RSI` to execute the specified shell script file:

```
...

activate_RSI(sup, sa)    /* used to activate an
  unstarted RSI */
struct subtransaction *sup;
struct sockaddr_in    *sa;    /* socket address */
{
    int  i;

    if (connect(sup->tcpFd, sa, sizeof(struct
      sockaddr_in)) < 0) {
        perror("connect");
        for (i = 0; i < sup->RSI_entry->RETRY;
          i++) {
            RsiRestart(sup->RSI_entry);
            if (connect(sup->tcpFd, sa,
              sizeof(struct sockaddr_in)) < 0)
                perror("connect");
            else
                break;
        }
        if (i >= sup->RSI_entry->TIMEOUT)
            Error_Handler("connect");
              /* error handling */
    }
}
RsiRestart(rsi_entry)  /* try to start an RSI remotely */
struct rsi_directory  *rsi_entry;
  /* the entry of RSI directory */
{
```

```
    int     fd, len;
    char    buf[BUFSIZE];
    char    *decrypted();
      /* routine to decrypt encrypted password */
    struct  servent  *sp;
      /* service entry */

    if ((sp = getservbyname("exec", "tcp")) == NULL)
      /* find the 'exec' port */
        Error_Handler("getservbyname");
          /* error handling */

    fd = rexec(&rsi_entry->hostName, sp->s_port,
      rsi_entry->USERID,
                        decrypted(rsi_entry->PASSWD),
                          rsi_entry->CMD, 0);
    if (fd < 0)
        Error_Handler("rexec");
    else {  /* grab and print output from the
      remote execution */
        while((len = read (fd, buf, sizeof(buf) - 1))
          > 0) {
            buf[len] = '0';
            OutPut ("rexec output: %s", buf);
        }
        close(fd);
    }
}
```

The RSI directory acts as another source from which a DFTM replica can obtain the TCP port for a specific RSI server, as this information can be specified in the appropriate entry in the RSI directory, as illustrated by the sample entry. The C program routine `connect_to_RSIs` for DFTM replicas described in Section 3.2 can therefore be modified as follows:

```
...

connect_to_RSIs() /* the routine for a DFTM replica
  to connect to RSI servers */
{
    ...
```

```
    /* for all subtransactions */
    for (sup = subtranQueue; sup != NULL,
      sup = sup->nextSubtrans) {
        /* obtaining the TCP port for the specified
          RSI server */
        if ((pse = getservbyname(sup->rsiName,
          "tcp")) == NULL)
            Warning ("No server %s in the table
                    /etc/services", name);
        else
            sup->tcpPort = ntohs (pse->s_port);

    ...
}
```

### *3.4 Communication Primitives for DFTM Replicas and RSIs*

Using TCP sockets as the communication medium between DFTM replicas and RSIs takes advantage of UNIX file I/O operations. However, these operations are designed for untyped byte streams and provide no means to distinguish among various network messages. If more than one network message has arrived at a TCP socket, a `read` system call might read in all available network messages, but if the network message is large, the `read` system call might read only a portion of the message. For this reason, `read` system calls cannot be used directly to receive network messages, and communication primitives are therefore needed to permit DFTM replicas and RSIs to distinguish among different InterBase messages. The following offers a simple solution:

| the length of the message(4 bytes) | the body of message(a command or data) |
|---|---|

Here, each InterBase message consists of two parts: the length of the message and the body of the message, which consists of a command or data and will be interpreted by the appropriate InterBase component. The C program routines `read_from_socket` and `write_to_socket` are thus developed to receive/send an InterBase message from/to a socket:

```
#include <malloc.h>
#include <sys/types.h>
#include <netinet/in.h>

char *read_from_socket(soc)
int   soc;      /* the file descriptor for a socket */
{
    u_long  nsize, size, len;
    char    *buf;       /* the memory buffer for
      storing network package */

    if (read (soc, &nsize, 4) != 4)
      /* get the size of the message */
        Error_Handler("read");
          /* error handling */
    else {
        size = ntohl(nsize);
          /* convert the size from network format
             to host format */
        if ((buf = malloc (size)) == NULL)
          /* get a free block of memory */
            Error_Handler("malloc'); ==
              /* error handling */

        for (i = 0; i < size; i += len)
          /* get the message */
            if ((len = read (soc, &buf[i], size - i))
              < 0)
                Error_Handler("read");
                  /* error handling */
    }

    return(buf);
}

void write_to_socket(soc, message)
int   soc;    /* the
le descriptor for a socket */
char *buf;    /* the memory buffer for storing
                 network package */
{
    u_long  i, j;
    i = strlen(message);     /* get the size of the
      InterBase package */
```

```
    j = htonl(i);            /* convert it from
      host format to network format */
    if (write(soc, &j, 4) != 4)
        Error_Handler("write"); /* error handling */
    else if (write (soc, message, i) != i)
        Error_Handler("write"); /* error handling */
}
```

As various formats may be used by different machines to present integers, the size of an InterBase message must be transformed from the corresponding host format to network format before it is sent to a remote socket. It must then be transformed from network format to the appropriate host format after it is received. These transformations allow the incompatability in integer representation among different computer systems to be resolved.

### *3.5 Handling Parallel Access to RSIs*

InterBase allows subtransactions within a global transaction to be executed in parallel whenever possible, improving global transaction response time and increasing InterBase throughput. This feature requires a DFTM replica to simultaneously handle multiple communication channels with several RSIs, which can be implemented through several approaches. A fairly inefficient solution would be to have a DFTM replica create a process to handle a communication channel. Still better would be to use the UNIX system call `select`, which can examine multiple I/O descriptors, such as sockets and returns to the caller, whenever there is an I/O descriptor ready for reading. After sending messages to the relevant remote RSIs, a DFTM replica can use the C program routine `WaitForEvents` to handle multiple communication channels simultaneously:

```
#include <sys/types.h>
#include <sys/time.h>

WaitForEvents(TimeOut)/* wait for an event to occur;
  if timeout, return TIMEOUT.  */
int     TimeOut;
{
struct subtransaction *sup;
fd_set fd_list; /* file descriptor set */
```

```
int     i;
struct timeval timeout; /* time out structure */

timeout.tv_sec = TimeOut; /* set timeout parameters */
timeout.tv_usec = 0;
FD_ZERO(&fd_list);
  /* initialize all descriptors for the list */

/* add those descriptors with pending replies */
for (sup = subtransQueue; sup != NULL;
  sup = sup->t_nextSubtrans) {
    sup->t_event = FALSE;   /* unmark all flags */
    if (sup->t_socket > 0 &&
      sup->t_status == WAIT_FOR_DATA)
        FD_SET(sup->t_socket, &fd_list);
          /* set a file descriptor */
}

/* wait for one of the pending replies to arrive */
if ((i = select (MAXFILEDES, &fd_list, 0, 0,
  &timeout)) < 0)
    Error_Handler("select"); /* error handling */
else if (i == 0) { /* Time Out */
    return(TIMEOUT);
}
    for (i = 0; i < MAXFILEDES; ++i)
      /* be very specific */
        if (FD_ISSET(i, &fd_list))
          /* there is a pending message */
            for (sup = subtransQueue;
              sup != NULL; sup = sup->t_nextSubtrans)
                if (sup->t_socket == i) {
                    sup->t_event = TRUE;
                      /* mark the flag */
                    break;
                }

    return (OK);
}
```

The t_event field of data structures for different subtransactions, generated by a successful call to the WaitForEvents routine, informs the DFTM replica as to the source of any RSI messages it has

received. It then receives and processes the messages in some predefined order, after which it may call the routine again or move on to another process.

### *3.6 Communication between an RSI Server, RSI Services, and DFTM Replicas*

As mentioned in Section 2, an RSI server coordinates the concurrent execution of DFTM replicas on its associated LSS; an RSI service, on the other hand, acts on behalf of a DFTM replica to execute a subtransaction of the DFTM replica on the LSS. An RSI therefore consists of a group of coordinating processes, with the RSI server as their parent. Since all RSI services are created by the appropriate RSI server, communication between the RSI server and its RSI services is easily established, often through a UNIX pipe. The situation is complicated, however, by the need of an RSI server to communicate with its RSI services and DFTM replicas simultaneously. As a polling strategy is excessively time-consuming, an approach similar to that described in Section 3.5 is preferable. The C program segment based on this strategy is as follows:

```
...

struct DFTM *headDFTM; /* the head of DFTM replica list */
struct RSIservice *headRSIS; /* the head of RSI service list */

main(argc, argv)    /* the main routine for an RSI server */
int   argc;
char *argv[];
{
    fd_set fd_list;             /* file descriptor set */
    struct timeval timeout;
    struct DFTM *dp;            /* DFTM replica */
    struct RSIservice *rp;      /* RSI service */
    int    i, fdConn, fdListen, si = sizeof(isa);
    struct sockaddr_in isa;       /* socket address */

    ...

    fdListen = get_listen_socket(); /* get a TCP socket */
```

```
for (;;) {  /* do forever */
    FD_ZERO(&fd_list);
      /* initialize the file descriptor set */
    FD_SET(fdListen, &fd_list);
      /* pending the TCP socket */

    for (rp = headRSIS; rp != NULL; rp = rp->nextRSIS)
        FD_SET (rp->pipe, &fd_list);
          /* pending the pipe of an RSI service */

    for (dp = headDFTM; dp != NULL; dp = dp->nextDFTM)
        FD_SET (dp->socket, &fd_list);
          /* pending the socket of a DFTM replica */

    timeout.tv_sec = TimeOut;    /* set timeout interval */
    timeout.tv_usec = 0;

    /* wait for messages come from one of the
       pending files */
    if ((i = select (MAXFILEDES, &fd_list, 0, 0,
      &timeout)) < 0)
        Error_Handler("select"); /* error handling */
    else if (i == 0) {
        TimeOut_Handle(); /* time out handling */
        continue;
    }

    if (FD_ISSET(fdListen, &fd_list)) { /* a message
      from a DFTM replica */
        /* accept a connection on the socket */
        if ((fdConn = accept(fdListen, &isa, &si)) < 0)
            Error_Handler("accept"); /* error handling */
        /* processing a connection request from a
          DFTM replica */
        process_DFTM_start(fdConn);
    }

     for (rp = headRSIS; rp != NULL; rp = rp->nextRSIS)
         if (FD_ISSET(rp->pipe, &fd_list))
             process_RSI_message(rp);
               /* process a message from an RSI service */

     for (dp = headDFTM; dp != NULL; dp = dp->nextDFTM)
         if (FD_ISSET(dp->socket, &fd_list))
```

```
                /* process a message from a
                   connected DFTM replica */
                process_DFTM_message(dp);
            ...
        }
        ...
}
```

An RSI service can also use a similar technique to communicate with its RSI server and its DFTM replica.

### *3.7 Communication between RSI Services and LSSs*

As mentioned in the previous subsection, an RSI service is used to execute a subtransaction for a DFTM on its associated LSS. The communication by which input is delivered to the LSS, output is collected, execution is monitored, the necessary format transformations are performed, falls within the responsibility of the RSI service. While communication among DFTM replicas, RSI servers, and RSI services is homogeneous, communication between RSI services and LSSs is heterogeneous, in that it is necessarily very-LSS specific, requiring detailed information about the workings of a particular LSS. This specificity rests upon the fact that such communication proceeds only through the standard interfaces provided by individual LSSs. For example, some LSSs, such as Sybase and Ingres DBMS, provide both query and programming interfaces, while others, such as the UNIX shell, provide only a command interface. Using either a programming interface or a query (command) interface has a different effect on the implementation of the RSI service.

Most Unix software systems allow applications to specify their sources of input data and command text and the destination of the output. For example, University Ingres lets users specify a command text and input data to be obtained from a text file, with the output to be redirected to another file. An RSI service can thus activate an execution of Ingres DBMS by combining the command text and input data into a file of its choosing, redirecting it to Ingres DBMS, and then redirecting the output to another file. When the execution of Ingres DBMS completes, all the output is in a file, available for

dispatch as needed. This UNIX property allows RSI services to use the query (or command) interface to an LSS. The C program routine RSI_Ingres_Exec provides an approach for such communication between University Ingres DMBS and its associated RSI service:

```
#include <stdio.h>
#include <sys/file.h>
#include <sys/wait.h>

...

/* a routine of University Ingres RSI service */
RSI_Ingres_Exec(outfile_des, cmdtext, input_data)
char *cmdtext, *input_data;
int   outfile_des;       /* output file descriptor */
{
    union wait  status;
    int   i, cpid, ifd, ofd, efd;
    char  *i_path, *o_path, *e_path;
    char   *MakeTemp(), *SetIngresInput();

    if ((cpid = fork()) == 0)  {  /* child process */
        e_path = MakeTemp ("ingres_e");
        o_path = MakeTemp ("ingres_o");
        i_path = SetIngresInput (cmdtext, input_data);
           /* set the input for ingres */

        if ((ifd = open(i_path, O_RDONLY)) < 0)
            Error_Handler("open");
        dup2(ifd, 0);              /* redirect stdin */

        if ((ofd = open(o_path, W_FLAGS, OUT_FILE_MODE)) < 0)
            Error_Handler("open");
        dup2(ofd, 1);              /* redirect stdout */

        if ((efd = open(e_path, W_FLAGS, OUT_FILE_MODE)) < 0)
            Error_Handler("open");
        dup2(efd, 2);              /* redirect stderr */

        for (i = 3; i < getdtablesize(); ++i)
          /* close ALL files from 3 */
            close(i);
```

```
        execl(INGRES, "ingres", INGRES_ARGS, INGRES_DB, 0);
          /* DO IT */
        Error_Handler("execl");
    }

    /* error happens during the execution of Ingres */
    if ((cpid = wait(&status)) == -1 || status.w_status != 0)
        return (process_ingres_error(e_path));
    else
        return (process_ingres_output(o_path, outfile_des));
}
```

However, although RSI services using query (or command) interfaces can be easily designed and implemented, there are several drawbacks. Because LSSs tend to output a combination of useful data and auxiliary explanatory data, the output tends to be difficult to interpret. Furthermore, error message arising from an error that occurs during the execution of an LSS will also pose interpretation problems to the RSI service. Finally, since different versions of an LSS may use different output and error message formats, an approach must be found to smooth this heterogeneity.

Unless the nature of the output and error message formats of an LSS is of no concern, it is preferable, for the above reasons, to use the programming interface of an LSS. Such an approach also permits an RSI service to use application programming facilities to support a larger number of applications. The C program routine `RSI_Sybase_Exec` provides a method for instituting such communication between Sybase DMBS and its associated RSI service:

```
#include <stdio.h>
#include <sybfront.h>
#include <sybdb.h>
#include <syberror.h>

/* Forward declarations of the error handler and message
  handler. */
int          err_handler(),  msg_handler();

LOGINREC     *login;     /* The login information. */
DBPROCESS    *dbproc;    /* The connection with the \*S. */
DBVARYCHAR    obuf[BUFSIZE];
```

```
...

RSI_Sybase_Exec(of_des, cmdtext, input_data) /* a routine
  of a Sybase RSI service */
char *cmdtext, *input_data;
int  of_des;          /* output file descriptor */
{
    char  *i_cmd, SetSybaseInput();
    int    i, j, ncol, ret_code;

    i_cmd = SetSybaseInput(cmdtext, input_data);
      /* set the input for sybase */
    dbcmd (dbproc, i_cmd);
    dbsqlexec(dbproc);     /* Send the command to Sybase
      for execution. */

    /* Process the output */
    while ((ret_code = dbresults(dbproc)) != NO_MORE_RESULTS)
        if (ret_code == SUCCEED) {
            ncol = dbnumcols(dbproc);
            for (j = 1; j <= ncol; ++j)
                dbbind(dbproc, j, VARYCHARBIND, 0, &obuf[j]);
                  /* Binding columns */

            while (dbnextrow(dbproc) != NO_MORE_ROWS) {
                for (i = 1; i <= ncol; ++i)
                  /* for each row of output */
                    process_sybase_output(obuf[i].str, of_des),
                      /* a column */
                process_sybase_output("\n", of_des);
                  /* end of a row */
            }
        }
        else
            process_sybase_error(ret_code);
    }
}

Login(passwd)
/*login to the Sybase DBMS, must be called prior to
  routine RSI_Sybase_Exec*/char *passwd;
{
    if (dbinit() == FAIL) /* Initialize DB-Library. */
        return (ERROR);
```

```
    /* Install user-supplied error-handling and
      message-handling routines.
     * The code for these is omitted from this example
       for conciseness.  */
    dberrhandle(err_handler);
    dbmsghandle(msg_handler);

    login = dblogin();              /* Get a LOGINREC.   */
    DBSETLPWD(login, passwd);
    DBSETLAPP(login, SYBASEAPPCMD);

    /* Get a DBPROCESS structure for communication
       with Sybase */
    if ((dbproc = dbopen(login, NULL)) == NULL)
        return (ERROR);

    return(OK);
}
```

### *3.8 System Component Crash Detection*

The TCP communication protocol makes possible reliable communication channels among InterBase components, narrowing concerns regarding reliability within InterBase to the reliability of InterBase components. Such InterBase components run on different computer systems, and it is unavoidable that errors of various types may arise during the execution of a global transaction. Some can be dealt with readily; for example, software errors can be avoided by careful debugging, while LSS error messages can be handled by carefully designed RSIs. On the other hand, many unpredictable errors may arise from the crash of some system component.

The many causes of a system component crash fall outside of this paper, only the topic of crash detection will be addressed here. Upon the detection of a crash, recovery may be undertaken through the appropriate action. For example, the procedures discussed in Section 3.3 can be used to reactivate crashed RSI servers; other approaches have been developed to recover InterBase from a crashed DFTM replica, a crashed RSI service, or a crashed LSS, as presented in [7].

While the timeout technique can provide a general indication regarding the crash of a system component, a combination of `rsh` and `ps` commands can be used in the UNIX environment to detect the running status of a particular process on a remote machine. The parameter concerned with process status can be substituted by a global transaction $G_i$ and the relevant RSI servers in determining the relative execution order of $G_i$. Therefore, if it is suspected that an InterBase component carrying process number 24426 on machine `ector.cs.purdue.edu` is down, a shell command can be issued to that machine for execution:

```
/usr/ucb/rsh -l interbase ector.cs.purdue.edu /bin/ps 24426
```

If the execution of this shell command returns the process status string which matches that earlier obtained, it can be concluded that this process is alive; otherwise, it is down. This shell command can be invoked within a C program routine using the C library routine `system` with redirected `stdout` and `stderr`.

If this command is executed on the machine where the suspected crashed process was run or is invoked using system call `rexec`, the `rsh` command is not necessary, although other parameters must be obtained to execute a `rexec` system call. For the previous example, the shell command can be modified to:

```
/bin/ps 24426
```

Unfortunately, the shell command `/bin/ps` may have a different synopsis on different UNIX platforms, appropriate minor changes may be required when InterBase is ported from one platform to another.

In order to use the `rsh` facility, file `.rhosts` for user `interbase` on all machines executing InterBase should have their `rsh` facility enabled for user `interbase` on all such machines. For example, if the InterBase system is installed on machines `ector.cs.purdue.edu`, `arthur.cs.purdue.edu`, `maggie.cs.purdue.edu`, `lisa.cs.purdue.edu`, ..., the contents of file `.rhosts` for user `interbase` on these machines should include the following statements:

```
ector.cs.purdue.edu interbase
arthur.cs.purdue.edu interbase
maggie.cs.purdue.edu interbase
lisa.cs.purdue.edu interbase
...
```

Any InterBase component can use the above strategy to detect a possibly crashed component and to take the appropriate action if that component is crashed. These actions may include reactivating the crashed component, or aborting itself, following the protocols described in [7].

## 4. *Implementing RSIs in Non-UNIX Environments*

At present, DFTM replicas are implemented only on UNIX platforms, while RSIs are implemented on UNIX, IBM CMS, and MS-DOS platforms. Because all RSIs follow the design principle illustrated in Section 2, we discuss only those issues particular to the implementation of RSIs on IBM CMS and MS-DOS platforms. These difficulties, which are not present in the implementation of RSIs on a UNIX platform, render the use of IBM CMS and MS-DOS platforms much more complicated and less efficient and flexible.

- IBM CMS and MS-DOS platforms are both single process environment; in that each user can run only one process at a time. The division of an RSI into an RSI server and an RSI service is therefore impossible on these platforms, and an RSI must be executed as a single process. As a consequence, at any given time, an RSI can either be involved in processing incoming requests from DFTM replicas or in executing a subtransaction, and subtransactions must thereby be executed sequentially. The execution of lengthy subtransactions may therefore cause incoming requests to be lost. It is preferable, as illustrated in Section 2, to develop the RSI server on a UNIX platform and the RSI service on a corresponding non-UNIX platform. The RSI server processes incoming requests and passes them in turn to the RSI service, which executes subtransactions for the incoming requests on the corresponding non-UNIX platform.
- As these two platforms do not allow one application program to directly invoke another, an RSI service must be developed as two separate application programs. The first one obtains an incoming request from the RSI server on a remote UNIX site

and prepares all the necessary parameters for the execution of the associated LSS. The second one processes the output from the execution of the LSS and sends the result to the associated DFTM replica. A batch execution file must therefore be developed for the execution of an RSI service. The following provides the logical execution sequence of this file:

```
do forever
   execute the first part of the RSI service that
     1) make network connection to the
       associated RSI server,
     2) obtain an incoming request from
       the RSI server,
     3) put comm. parameters to the appropriate
       DFTM replica into file LSS.DFTM,
     4) put transformed execution params from
       the incoming request into file LSS.EXEC;

   execute the associated LSS using file LSS.EXEC as
         the execution parameter and put the
           output into the file LSS.OUT;

   execute the second part of the RSI service that
     1) use the files LSS.OUT and LSS.DFTM
       as input parameters,
     2) make network connection to the associated
       DFTM replica,
     3) process the output data in file LSS.OUT,
     4) send the result to the associated DFTM
       replica.
enddo
```

- IBM mainframes use EBCDIC character set, which is incompatible with the ASCII character set used by UNIX platforms. The characters in InterBase messages must be transformed to the appropriate character format when they are processed.
- These two platforms do not provide mechanisms similar to the `rexec` and `rsh` facilities of the UNIX platform. It is therefore impossible to remote-activate an RSI service and to remote-detect the crash of an RSI service on either platform. Although the timeout technique can provide an indication of the

crash of an RSI service, only the human operator can reactivate a crashed RSI service, greatly reducing flexibility.

- These two platforms provide no built-in TCP/IP protocol. A TCP/IP simulator must therefore be used, and a communication protocol compatible with UNIX file I/O operations based on the TCP/IP simulator must be developed. For example, in the case of an IBM mainframe, a TCP/IP simulator, knet, can be used as the basis of a communication protocol compatible with UNIX file I/O operations.

## 5. *Conclusion and Future Work*

UNIX network facilities provide the necessary foundation to construct mechanisms for cooperative interactions among component database systems and auxiliary system components in heterogeneous multidatabase systems. Implementation details of such mechanisms have been presented in this paper. Such cooperative interactions are much more difficult to achieve in IBM-CMS and MS-DOS platforms. Although better solutions may be developed using advanced UNIX facilities such as RPC and MACH threads[3], the method presented here does offer an innovative, portable, and effective practical approach to the institution of cooperative mechanisms among system components of heterogeneous multidatabase systems. The practicality of this solution has been demonstrated through a trial implementation of InterBase in the Quality Control Department of Bell Northern Research, Inc. for more than two years.

3. Inputs from several sockets can be handled simultaneously, permitting the most efficient use of sockets. Their low-level implementation, however, makes the program difficult to understand and to debug. Because the format of RPC calls resemble that of local procedure calls, using RPC facility avoids the difficulties caused by sockets. Current version of RPC is, however, restricted by the fact that each RPC call is blocked until a value is returned. Although parallel execution of several RPC calls can be implemented using UNIX processes, it is however time-consuming and requires unnecessary inter-process communication. If MACH threads can be used instead of UNIX processes, these extra-costs can be reduced to minimum. Moreover, using MACH threads has the advantage of reducing the creation cost of RSI services and communication cost between an RSI server and its associated RSI services. However, since several threads are allowed to share a common memory, mutual exclusion among these threads must be addressed to avoid mutual interferences. Furthermore, RPC and MACH threads may not be available to some UNIX machines and there are incompatible versions of RPCs for various platforms.

InterBase is currently based on a UNIX platform; porting InterBase to other computer platforms will be the subject of future research. Additional LSSs will be incorporated into InterBase, permitting the support of a larger number of applications and of a larger community of users.

At present InterBase uses IPL, a low-level language, as its programming interface. This will be extended in the future with an interactive user-friendly graphical interface, which will render the syntax of individual local applications highly transparent to the user. An initial version of this graphical interface has been demonstrated to a group of industrial affiliates. An object-oriented user interface, also under development, will provide an object-oriented SQL interface to InterBase. A detailed performance evaluation of the InterBase System is also planned as a part of our future investigations.

## *References*


[1] R. Ahmed, P. De Smedt, W. Du, W. Kent, M. A. Ketabchi, W. A. Litwin, A. Rafii, and M.C. Shan. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer,* 24(12):19-27, December 1991.

[2] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceedings of the 18th VLDB conference,* Vancouver, Canada, August 1992.

[3] V. Belcastro et. al. An overview of the distributed query system dqs. In *Computer Science,* pages 170–189, Vol. 303, Springer-Verlag, New York, 1988.

[4] D. A. Bell, J. B. Grimson, and D. H. O. Ling. EDDS—A System to Harmonize Access to Heterogeneous Databases on Distributed Micros and Mainframes. *Information and Software Technology,* 29(7), 1993.

[5] Y. Breitbart, P. L. Olson, and G. R. Thompson. Database Integration in a Distributed Heterogeneous Database Systems. In *Proceedings of the 2nd International Conference on Data Engineering,* pages 301–310, Los Angeles, CA, February 1986.

[6] A. Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A transaction model for active distributed object systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[7] Omran A. Bukhres, Jiansan Chen, Weimin Du, Ahmed K. Elmagarmid, and Robert Pezzoli. InterBase : An Execution Environment for Global Applications over Distributed, Heterogeneous, and Autonomous Software Systems. *IEEE Computer,* August 1993.

[8] Jiansan Chen, Omran A. Bukhres, and Ahmed K. Elmagarmid. IPL: A Multidatabase Transaction Specification Language. In *Proc. of the 13th International Conference on Distributed Computing Systems,* Pittsburgh, PA, May 1993.

[9] C. W. Chung. DATAPLEX: An Access to Heterogeneous Distributed Databases. *Communications of the ACM,* 33(1):70–80, January 1990.

[10] Douglas E. Comer. *Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture,* volume I. Prentice Hall, Inc, second edition, 1991.

[11] C. Esculier. The Sirius-Delta Architecture: A Framework for Cooperating Database Systems. *Computer Networks,* 8(1), 1984.

[12] P. M. Strocker et al. Proteus: A Heterogeneous Distributed Database Project. *Cambridge Univ. Press,* 1984.

[13] S. Spaccapietra et al. Scoop—A System for Cooperation between Existing Heterogeneous Distributed Databases and Programs. *Database Engineering,* 5(4), 1982.

[14] W. Staniszkis et al. Architecture of the Network Data Management System. In *Proceedings of the 3rd Int'l Seminar on Distributed Data Sharing Systems,* Amsterdam, North-Holland, 1985.

[15] Z. Brzezinski et al. Unibase—An Integrated Access to Databases. In *Proc. of the 10th Int'l VLDB Conference,* San Mateo, CA, 1984.

[16] B. Holtkamp. Preserving Autonomy in a Heterogeneous Multidatabase Systems. In *Proc. of the 12th Int'l Computer Software Applications Conf.-Compsac 88,* Los Alamitos, CA, 1988.

[17] E. Kuehn and Y. Ludwig. Vip-mdbs: A logic multidatabase system. In *Proceedings of the International Symposium on Database in Parallel and Distributed Systems,* Austin, Texas, December 1988.

[18] T. Landers and R. Rosenberg. An Overview of Multibase. In H. Schneider, editor, *Distributed Data Systems.* North-Holland, 1982.

[19] W. Litwin. An Overview of the Multidatabase System MRDSM. In *Proceedings of the ACM Annual Conference,* pages 524–533, Denver, Colorado, 1985.

[20] X. Liu, J. Chen, and R. Pezolli. The InterBaseView Graphical User Interface. Technical Report SERC-TR-126-P, Department of Computer Sciences, Purdue University, November 1992.

[21] A. O. Omololu, N. J. Fiddian, and W. A. Gray. Confederated Database Management Systems. In *Proc. of the 7th British Nat'l Database Conference.* Cambridge Univ. Press, 1989.

[22] Relational Technology, Inc. *Ingres,* 1986.

[23] M. Rusinkiewicz et. al. Omnibase: design and implementation of a multidatabase system. *Distributed Processing TC Newsletter, IEEE Computer Society,* 10(2):20–28, 1988.

[24] Sybase, Inc. *Transact-SQL User's Guide,* 1987.

[25] M. Templeton, E. Lund, and P. Ward. Pragmatics of Access Control in Mermaid. *IEEE Data Engineering Bulletin,* 10(3):33–38, 1987.

[26] D. Woelk. Using Carnot for Enterprise Information Integration (Synopsis). In *Proceedings of the Second International Conference on Parallel*

*and Distributed Information Systems,* pages 133–136, San Diego, CA, January 1993.

[27] A. Wolski. Linda: A System for Loosely Integrated Databases. In *Proc. of the 5th Int'l Data Engineering Conference,* Los Alamitos, CA, 1989.