FACTA UNIVERSITATIS (NIŠ)
Series: Electronics and Energetics vol. 9, No.2 (1996), 149–172

THE DESIGN AND THE IMPLEMENTATION OF A DOSTP - A DISTRIBUTED OBJECT ORIENTED SYSTEM BASED ON TRANSACTIONS PROCESSING

Carmen Musataescu and Dumitru Dan Burdescu

Abstract. This paper presents an experimental object oriented system that provides operating system level support for distributed transactions that operate on share objects. Transactions are a common construct that distributed programming environments provide in order to maintain the consistency of distributed information in the presence of partial failure and concurrency. The goal of constructing a transactions support in a distributed object oriented system is to make transactions available as a fundamental programming constructs for reliable distributed computing; on the other hand, the distributed environment can be used to exploit the parallelism in some computation within transactions. This paper presents the basic components of a distributed object oriented system based on transactions processing and its implementation on a local network of DEC Stations 3000 under OSF/1 vers 1.32 and Zenith PC under Linux 1.2.8 and on a NOVELL network under NetWare 3.11. Basic transport–level communication is performed by TCP/IP, UDP/IP via an Ethernet and by IPX, respectively.

1. Introduction

The potential benefits of distributed processing are now recognized. Therefore, there is a great interest in general-purpose methodologies and techniques that simplify the construction of efficient and robust distributed applications. Now, **atomic transactions** are quite a familiar paradigm for the construction of reliable distributed applications [22].

Transactions are originally developed for database management systems, to aid in maintaining consistency constrains on stored data. Database management systems are not the only ones that must assure the consistency of

Manuscript received May 21, 1996.

The authors are with Department of Computer Science, University of Craiova, Str. Tehnicii No 5, Craiova 1100 Dolj, Romania.

stored data despite failures and concurrency. Argus, TABS/Camelot supports the nested transaction scheme [22]. Eden supports a variation of the nested transaction scheme in which an action must be explicitely committed by the user [22]. Transactions are also available in the Clouds distributed operating system.

Although operating systems support for transactions and atomicity is beginning to appear, the desirability of supporting transactions and atomicity has not yet been generally accepted [6].

This paper describes the design and implementation of a distributed object oriented system, called DOSTP, based on transactions processing, developed at the University of Craiova, Department of Computers, on a UNIX local area network and on a NOVELL NetWare 3.11 network.

2. Objects and Transactions in DOSTP System

In DOSTP system major components are: the objects, the transactions and the application processes. The application processes invoke operations on objects by using atomic transactions. In an atomic transaction may appear any number of objects located locally or remotely.

An atomic transaction begins with a **BeginTransaction** function which is used by the DOSTP system to allocate necessary resources to transaction. Any operation of an object may be executed in either synchronous or asynchronous mode. If an operation returns an error code, the application process must request that the atomic transaction be aborted.

A transaction can be committed only if all operations involved in that transaction complete successfully. Prior to transaction commit, the application process may request to wait until all the asynchronous initiated operations are (successfully or unsuccessfully) completed.

In DOSTP, an **object** (persistent object) is a container for data; each object has a type that defines a set of primitive operations. An **Object Manager** encapsulates its data objects and the operations that manipulate it. The operations on objects are performed only by the Object Managers that are encapsulated in processes. Object Managers receive method invocations via a request message.

Objects are active entities, doing work only when their methods are invoked. Object Managers are called in these case **server processes**. An application process may begin many asynchronous operations on objects; these operations are performed in parallel by the Object Managers.

The Object Managers encapsulate the state of objects as well as their behavior that collectively describes the notion of active objects. The state contains the data structures of the objects involved in the requests and the synchronization requirements necessary to ensure that the activities of multiple operations that invoke the same object do not conflict or interfere with one another. Object Managers provide the means to maintain the consistency of objects. We developed for DOSTP two mechanisms for synchronization: one uses a pessimistic synchronization scheme and the other uses an optimistic synchronization scheme. The pessimistic scheme uses Read/Write locks. However, Object Managers can increase concurrency by exploiting the semantics of operations. Object Managers may use type-specific locking, a mechanism which employs type-specific properties of objects to recognize when certain operations such as concurrent write operations are not conflictual. In the pessimistic scheme (to synchronize access to the objects) an operation that invokes an object is temporary suspended if it will interfere with other operation that is currently being serviced by the Object Manager. The suspended operation will be resumed only when all transactions containing the conflicting operations commit.

When an Object Manager performs an operation, it then sends a response message containing the result. From the point of view of an Object Manager there may be many uncommitted transactions active at any given time.

In order to provide failure atomicity and persistent objects, Object Managers have also other responsibilities in committing and aborting transactions and also in recoveries of transactions.

DOSTP system uses a log-based recovery algorithm. In a log-scheme, whenever a persistent object is modified, information about changes are recorded in secondary storage. The information recorded in a log file serve two purposes: to undo the updates of transactions that abort and to redo the transactions that commit prior to a crash but whose updates were lost or destroyed by the crash. DOSTP uses a write-ahead logging algorithm. In order to ensure the atomicity of transactions, these algorithms require that undo information must be forced on log before the object modified by the current update operation is overwritten on persistent storage and redo information must be forced before a transaction can commit[6].

The write-ahead log algorithm implemented in DOSTP is based on value logging. The Object Managers log the old and new value of updated objects whenever they update a persistent object. The advantages of a write ahead logging are recognized in many paper [6][7]: reduce I/O activity both at transaction commit time and during recovery, when the log is typically scanned sequentially, the potential for higher concurrency, the potential for an easier management of the objects.

Distribution in DOSTP is largely transparent: a programmer must never know where an Object Manager resides. In fact, invoking an operation on an object residing on a remote node has exactly the same effect as if it performed locally, except of course for a poor performance. Therefore, applications can be developed and debugging in a single node environment and then moved to the network.

In DOSTP system, Object Managers can execute in parallel multiple transactions.

3. The basic elements of DOSTP

DOSTP system is based on seven types of processes located on the top of a multitasking kernel. One type of these processes is user-programmable. This is the Object Manager briefly presented previously. The components of DOSTP are shown in figure 1 and are briefly described below:

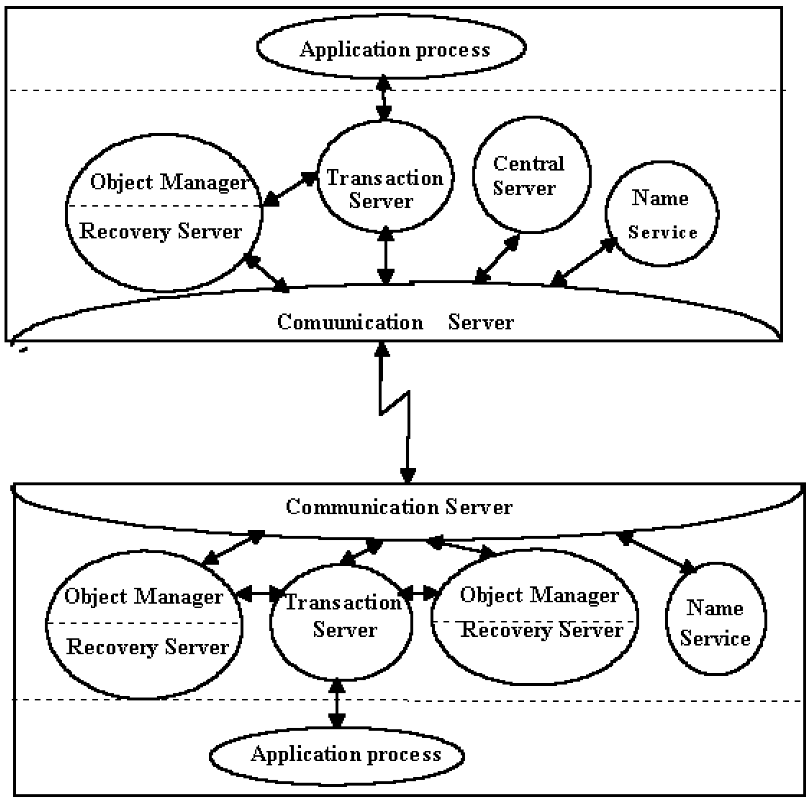

Figure 1. Two nodes of DOSTP system.

1. The **Communication Server** is a single process in each node of DOSTP system. Its purpose is to handle communications between dif-

ferent nodes. An application process operates like in a local process environment. Therefore, communication between DOSTP nodes involves inter-node messages and inter-process communication. The Communication Server provides node to node message forwarding. The communication between a pair of processes located on different nodes is achieved by interposing two Communication Servers (one in each node) between the sender process and the receiver process.

2. The **Name Service** is a single process in each node of DOSTP system. Its purpose is to filter the objects: local or remote.

Because of a high degree of location transparency is supported, the Name Service will have to maintain data structures of its own. The users need not to know where objects are stored. The user merely names the object (identify its type and its name) and the Name Service finds the correspondent Object Manager.

3. The **Transaction Server** is a single process in each node of DOSTP system. The Transaction Server receives and treats all the requests of BeginTransaction, EndTransaction and AbortTransaction on behalf of local or remote transaction. The Transaction Server also records information about the nodes involved in a distributed transaction.

4. The **Central Server** is a single process in the system; its role is to assign identifiers and priorities to transactions.

5. The **Object Manager** is the process that coordinates and stores many objects of the same type. These are persistent objects stored in one or more corresponding files. The Object Manager synchronizes the accesses to the same object using type-specific locking scheme.

6. The **Recovery Server** is a single process for each Object Manager. The Recovery Server coordinates the process of recovery after a transaction aborts. The Recovery Server has also functions during normal execution of a transaction. The Recovery Server receives the records to be written in the log file from its correspondent Object Manager and from Transaction Manager and writes these records in the log file.

7. The **Application processes** are written by the users and invoke transactions and operations on objects. In each DOSTP node there may be many Application processes.

4. UNIX Implementation

At the time of this writing, the DOSTP system was implemented on a local network of DEC Stations 3000 under OSF/1 vers 1.32 and Zenith PC under Linux 1.2.8. Basic transport–level communication is performed by TCP/IP via an Ethernet.

In order to connect application processes, DOSTP (the Transaction Server) has in each node a FIFO queue whose name is public. Any application process that wants to be connected to DOSTP, must at first put a connect request message in this queue (a **BeginTransaction** message). Prior to transaction begin, each application process must create a FIFO queue in which DOSTP will send the reply messages. DOSTP knows that the name of this FIFO queue is the name of the Application process (written in the BeginTransaction message) followed by the suffix .FIFO. DOSTP answers to a BeginTransaction message with a reply message whose structure is shown below:

```
typdef struct{
  int type;
  int ErrorCode;
  int Shmid,Semid;
} BEGIN_TRANSACTION_ACKNOWLEDGE;
```

In a BeginTransaction acknowledge message, the shmid field contains the identifier of a shared memory segment. DOSTP (the Transaction Server) creates this shared memory segment in order to receive and transmit all the messages from/to application process. In order to ensure the synchronization of all processes that will have access to the shared memory segment, DOSTP creates and initializes a set of semaphores and returns its identifier, (in semid field) to the Application process.

Application processes are programmed with the aid of some library functions that contains the code for doing the connection to DOSTP system, for beginning, committing and aborting a transaction, for invoking operations on objects, for waiting the termination of an asynchronous operation, and so forth.

In order to connect it itself to DOSTP system, an Application process must execute the **Init(ApplicationName)** and **BeginTransaction()** functions.

It is the Transaction Server's responsibility to receive and treat the connection requests (the BeginTransaction messages); it then replies with an **BeginTransactionAck** message placed in the Application process' FIFO queue. The Transaction Server is a daemon process which creates a child process (fork, exec) for each BeginTransaction message received and returns then an identifier for that transaction. It is newly created process' responsibility (called **Local Transaction Server**), to coordinate the transaction.

The Local Transaction Server executes at the initiation of a transaction a sequence of operations briefly described below:

```
void Init (char *QueueName)
{
... /* connect the handlers for SIG_CHLD, SIG_ALRM */
... /* connection to the application queue */
... /* creates the shared memory segment */
... /* it tries to attach to the shared memory segment */
... /* creates the set of semaphores */
... /* request to the parent process (Transaction Server) to reserve an iden-
tifier for transaction */
... /* build the reply message */
... /* sends the reply message to the application */
}
```

After a transaction begins, the Application process invokes an operation on an object via a request message placed in the shared memory segment and receives the expected response messages in the shared memory segment too. In fact, many messages between DOSTP processes are placed in the shared memory segment. The "producer process" will insert the message in the shared memory segment informing the "consumer process" by executing an UP operation on a semaphore. The "consumer process", blocked at this semaphore, will receive this message in its own queue, placed in the shared memory segment.

The operation requests from Application processes to Object Managers are delivered to the Transaction Server (in fact, Local Transaction Server). After some routine processing, these requests are forwarded to Communication Server. The Communication Server, using Name Service, knows the location of the objects in the network. If the object is local, and the request is the first request for that object, the Communication Server sends a message to the Object Manager in order to obtain a connection. The Object Manager creates a child process that tries to attach itself to the shared memory segment. From this moment, the child process accepts messages requesting what operations be performed and returns results and error codes in reply messages placed directly in shared memory segment.

When the Object Manager resides on a node different from that of the Application process, the requests to the Object Manager are forwarded by the Communication Server at each node.

The Transaction Server from the Application process' node records when a new node joins the transaction (in that case, the transaction has spread to another node).

From the point of view of the Object Manager, there is only one way to receive the requests and sends the results: by using a shared memory segment (even if the Application process resides on a node different from that of the Object Manager).

The structure of a shared memory segment is shown below:

```
typedef struct
{
  PACKET SlotTable[MAX_SLOT];/* packets* /
  int FreeList[MAX_SLOT];/* free slot list */
  int IndexFreeList;/* index in the free slot list */
  int AppQueue[MAX_SLOT];/* message queue for an application process */
  int IndexAppQueue;/* index in an application process queue */
  int TransServerQueue[MAX_SLOT];/* message queue for Transaction Ser-
ver */
  int IndexTransServerQueue;/* index in Transaction Server queue */
  int CommServerQueue[MAX_SLOT];/* message queue for Communication
Manager */
  int IndexCommServerQueue;/* index in Communication Manager queue */
  int ObjectManagerQueue[MAX_OBJ][MAX_SLOT]; /* arrays of queues for
Object Managers */
  int IndexObjectManagerQueue[MAX_OBJ];/* arrays of indexes in the queu-
es for Object Managers */
}SHM;
```

The SlotTable collect all the messages, called packets, between the Application process and the DOSTP modules. The structure of a packet is shown below:

```
typedef struct
   {
  int type;/* packey type */
  int ErrorCode;/* error code */
  int TransIdentifier;/* transaction identifier */
  int ObjectIdentifier;/* object identifier */
  long SeqNumber;/* the packet's number in this transaction */
  int AppSlot;/* used by the Application */
  long AppSlotUsage;/* used by the Application */
  int InternId;/* used by the Transaction Server */
  char ObjectName[OBJECT_NAME_SIZE];/* the name of the object */
   REQUEST text;/* the request or result message */
   } PACKET;
```

The positions of the packets in the SlotTable are pointed by arrays of indexes: AppQueues [MAX_SLOT], TransServerQueue [MAX_SLOT], CommServerQueue [MAX_SLOT], ObjectManagerQueue [MAX_OBJ] [MAX_SLOT]. The entire shared memory segment is protected by a semaphore, the S_SHM semaphore. Messages may be sent and received from the shared memory segment using semaphores for synchronization. These semaphores are allocated either statically at the beginning of the transaction, or dynamically during the access to objects.

The Object Managers return the results and the exceptions to Application processes via shared memory segment. If the Object Manager resides on a node different from that of the application process, the results are forwarded by the Communication Server.

Every Object Manager must be recorded in DOSTP system. At the connection time, an Object Manager must be registered by the DOSTP system. Therefore, every new Object Manager begins its execution by sending an authentication message to the local Communication Server. The Communication Server sends a message to the Central Server, who is responsible to forward this message to every Name Service. After sending an authentication request, the Object Manager will receive a reply message in its FIFO queue.

The connection of an Object Manager to the DOSTP system is ready when the Object Manager receives the reply message to its authentication request.

After connection, the Object Manager can receive request messages; the first message may be only of the BEGIN_TRANSACTION_OBJECT type. This type of message stores the identifier of the shared memory segment, the number of the semaphore allocated for the synchronization in the access to the shared memory segment, the identifier of the queue reserved for the Object Manager in the shared memory segment and the name of the Application process.

The structure of a BEGIN_TRANSACTION_OBJECT message received by an Object Manager is shown below:

```
typedef struct
{
  int type;
  int ErrorCode;
  int shmid; /* shared memory identifier */
  int semid; /* semaphore identifier */
  int SemNumber;
  int QueueNumber;
  char AppName;
}BEGIN_TRANSACTION_OBJECT;
```

The ObjectManager creates a child process for every new transaction. Because the Object Managers are user programmable modules, there is a set of available library functions for connection to the DOSTP system, for receiving and sending messages.

In order to be connected to the DOSTP system, Object Managers must call the Init function; the code of an Object Manager is briefly described below:

```
int Input(arguments ...)
{
n=read(fdPipe, &packet, sizeof(packet);
if (!authentication)
  if (packet.type == AUTHENTICATION)
    if (packet.ErrorCode == MY_OK) authentication = 1;
    else {
    .../* the connection of the Object Manager to DOSTP system failed */
    }
  else {
  .../* a packet was received before the connection to DOSTP system */
  .../* ignore this packet */
  }
else {
  switch(packet.type){
  case BEGIN_TRANSACTION:
    TransNumber++;
    .../* creates a child process who will coordinate this transaction */
    break;
  case ACCEPTED_TRANSACTION:
    .../* the child process accepts the transaction */
    .../* it succeeded to attache itself to the shared memory segment */
    .../* the transaction is inserted in a list of transactions */
    break;
  case NOT_ACCEPTED_TRANSACTION:
    .../*the child process failed to connect itself to the shared memory segment
*/
    break;
  case END_TRANSACTION:
    .../* transaction ended successfully */
    .../* deletes the transaction from the list of transactions */
    break;
  case ABORT_TRANSACTION:
    .../* transaction failed */
    .../* deletes the transaction from the list of transactions */
    break;
  ...
  default:
    .../* unknown packet type */
    break;
  }
  /* end switch */
}
}
```

The child process created by the Object Manager will receive the BEGIN-_TRANSACTION_OBJECT packet whose fields identify the shared memory segment and will try to attach itself to this shared memory segment. If the child process succeeds, it then will inform its parent that the transaction is accepted; otherwise the parent receives the message NOT_ACCEPTED_TR-ANSACTION. The child process can now receive packets from the application process in the shared memory segment. Therefore, it blocks itself until it receives a packet. The child process executes the main loop shown below:

```
void ChildProcess(void){
.../*initialize*/
while (JobToDo) {
  WaitForMessage (&packet);
  TreatMessage (&packet);
}
... /* dettach from the allocated resources */
}
```

When the Application process sends the EndTransaction message to the Transaction Server (in fact, to the Local Transaction Server) informing that the transaction wants to commit, the commit processing may begin. The two–phase commit protocol is coordinated by the Local Transaction Server. The Local Transaction Server knows if remote sites are involved; it obtains from a local list the names of the child nodes and sends the PrepareToCommit message to all the Object Managers. An Object Manager (in fact, its child process who executes this transaction) treats a PrepareToCommit message with the code briefly described below:

```
void PrepareToCommit (PACKET* pPacket)
{
  ... /* check the request message */
  ... /* suspend receiving any new operation requests for that transaction */
  ... /* wait until all operations for that transaction are finished */
  ... /* for optimistic version only: check if the modifications can be written
*/
  ... /* send back a PrepareToCommitAck message */
}
```

When the Local Transaction Server, who is the coordinator of the transaction is informed that all remote nodes have prepared, it knows that now it can commit the transaction.

The Object Manager treats a Commit Transaction and an Abort Transaction message with the code briefly described below:

```
void CommitTransaction (PACKET *pPacket)
{
  ... /* check the syntax of the request message */
  ... /* for optimistic version only: install the modifications */
  ... /* sends a CommitTransaction record to the Recovery server */
  ... /* drops the locks that were holding on behalf of this transaction */
  ... /* sends back a CommittAck message */
}

void AbortTransaction(PACKET *pPacket)
{
  .../* check the syntax of the request message */
  .../* sends an Abort message to the Recovery Server */
/* then, the Recovery Server logs an Abort record in the log file */
  .../*for pessimistic version only:
sends a RollBack message to the Recovery Server in order to undo the changes
that this transaction has made
:loop: the Recovery Server finds the most recent modified record of the abort-
ing transaction; this record is sent to the Object Manager who undo this op-
eration. The Recovery Server repeats this process for each of the aborting
transaction's modified record. When the roll back process is finished, the
Recovery Server sends a FinishedRollBack message */
  ... /* drops the locks that were holding on behalf of this transaction */
  ... /* inform the parent process that the transaction aborted */
}
```

The library functions available to an Application process are briefly described below:

```
  int Init(char *AppName);/* connection to DOSTP system */
  int End(void);/* finish connection to DOSTP */
  int BeginTransaction(void);/* the Application process begins a transaction
and request a shared memory segment for communication */
  int EndTransaction(int);/* a succesfully ended transaction */
  int AbortTransaction(int);/* an unsuccessfully ended transaction */
  int AsyncRequest(char *ObjectManagerName, void *request, void *answer,
int *pErrorCode, int *ResponseTime) /* an asynchronous request to the Ob-
ject Manager */
  int Synchronize(void);/* wait for the results of previous asynchronous re-
quests */
```

The Application process is not forced to wait for the completion of an operation; in this case, the Application process must use asynchronous requests and may initiate many concurrent operation.

5. NOVELL Implementation

In a Novell NetWare 3.11 Operating System, we developed a multitasking kernel in each node. The structure of a task is the following:

```
class Task
{
protected:
  char Name[8];//the name of the task
  char *stack;//pointer to the stack
  struct TCB *taskTCB;//pointer to a TCB structure
public:
  Task(char*, WIN*, unsigned=1024);//constructor
  Task();
  T̃ask();
  virtual void farTaskMain()=0;
  void EndTask();
  struct TCB *getTCB(){return taskTCB;}
  char *Read(CHANNEL*);
  void Write(CHANNEL*,char*);
};
```

The kernel allocates for every task a Task Control Block and a stack. A Task Control Block has the following structure:

```
struct TCB
{
Task *t;
unsigned r_ss; // registers
unsigned r_sp;
unsigned r_bp;
State TaskState; // task state
ulong ticks;
WIN window;
task window information
};
```

The communication between tasks is performed using a channel object; its definition is described below:

```
class CHANNEL
{
  protected:
  SEMAPHORE s; // local semaphore for mutual exclusion
public:
  CHANNEL:(unsigned char =256);//constructor
  C̃HANNEL();
  int write(const char* ...);
  int read(const char* ...);
  char *buffer;
};
```

The structure of a local semaphore is presented bellow:

```
class SEMAPHORE
{
private:
  int value;
protected:
  TCB *BlockedTasks[MAX_TASKS]; //list of blocked tasks
public:
  SEMAPHORE():value(1){};
  SEMAPHORE(int val):value(val){};
  ˜SEMAPHORE();
  void P(TCB *);
  void V(void);
};
```

The task hierarchy in DOSTP is described in figure 2.

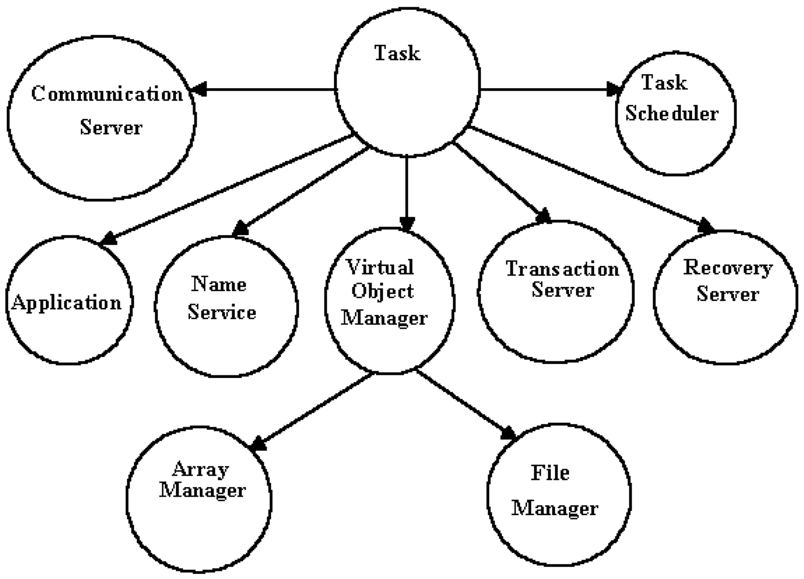

Figure 2. The task ierarchy in DOSTP.

At the time of this writing we have developed many Object Managers and executed some simple transactions. The most important Object Manager that we developed is a File Manager.

The structure of a Virtual Object Manager is shown below:

```
class VirtualObjectManager: public Task
{
public:
  VirtualObjectManager (char*,WIN*, unsigned,CHANNEL*,CHANNEL*,
CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*); //
constructor; first, it calls Task class constructor and then iniatilizes seven
channels for communication with other DOSTP tasks
  ṼirtualObjectManager;//class destructor
  virtual void TaskMain() {};//task main function
  virtual int Read(char, char*);//virtual functions can be overriden in a class
and thus adapted to fit new situations
  virtual int Write(char,char*);
  virtual int Delete(char,char*);
  virtual int Modify(char,char*);
  virtual int IsConflict(char, char, char*, int, char, char, char*, int)=0; //ver-
ify the concurrence
  virtual int Switch(char, char, char)=0;// select the operation
  virtual void MsgToNameService()=0;// send initial information to the Na-
me Service
  request *req;
  char message[MES_LENGTH];
  CHANNEL *channel_om_cs, *channel_cs_om;
  CHANNEL *channel_om_ts, *channel_ts_om;
  CHANNEL *channel_om_nm;
  CHANNEL *channel_om_rs, *channel_rs_om;
protected:
  int exRqNmb, TrNmb;// number of unfinished requests and number of
transactions
  executed execReq[MAXREQ*NRIDTR];//array of finished requests
  status BlkReqQs[NRIDTR];//array of waiting requests
  void IdTransSearch(int);//search in BlkReqQs
  void WaitingQueueInsert(char,int);//insert a request in the waiting queue
  void WaitingQueueDelete(int);// delete all the waiting requests of a trans-
action
  void ExecuteRequest(int);//execute or enqueue the request
  void ExecuteQueue(int);//execute all the requests from a transaction queue
};
```

The derivation of VirtualObjectManager from the Task class allows VirtualObjectManager to inherit the Task class members.

Each Object Manager has allocated seven channels for communication with other modules of DOSTP. These channels are reserved for the following communications: Object Manager to/from Communication Server (om_cs, cs_om) , Object Manager to/from Transaction Server (om_ts, ts_om), Object Manager to/from Recovery Server (om_rs, rs_om), Object Manager to Name Service (om_nm).

An Object Manager receives two types of operation requests: synchronous and asynchronous. When an application calls a synchronous operation, it then blocks itself until the Object Manager performs the operation. DOSTP system provides also asynchronous operations and uses a function called Synchronize(), to notify that the operation was performed. The Object Manager keeps the data structures necessary to execute these two types of operations.

Execute Request function is called for each operation request, either locally or remotely. ExecuteRequest function is called also by the ExecuteQueue function in order to initiate the execution of the next blocked operation. ExecuteQueue function is called whenever a transaction commits or aborts. During normal execution of an operation on behalf of a transaction, the Object Manager can block another operation on the same object on behalf on other transaction; when the first transaction commits or aborts, Object Manager must unblock the second transaction. This process is performed using the following functions: IsConflict function, in order to detect a conflict between operations from different transactions, ExecuteQueue function, in order to unblock operation requests and WaitingQueueInsert, in order to insert an operation request in the array of blocked requests. WaitingQueueDelete function is called whenever a transaction aborts in order to delete any blocked operation on behalf of that transaction. The reason for this is the following: if two applications use the same object in simultaneous transactions, the first one to finish wins and makes all its changes and the second, after a time out, is aborted and all its blocked operations must be deleted. If a timer goes off before the reply comes back for an operation (this is the case of a deadlock) then the Communication Server sends an Abort Transaction message to the Transaction Server to abort the transaction. Then the second is aborted. The second application is effectively rolled back by the Recovery Server to the start of its transaction, so no damage has been done to any of its objects. In order to know what operation must be unblocked when a transaction aborts, the Object Manager keeps an array or records, BlkReqQs[NRIDTR]. An element of this array has the structure shown below:

```
struct status
{
  int InUse;//used or unused entry
  int TransId;//transaction identifier
  char TransState;//the state of that transaction
  wqueue *qHead;//begining of the blocked operations queue for that trans-
action
};
```

For every unfinished transaction, the Object Manager keeps a queue of blocked operation requests. If that transaction is aborted (time–out or software abort), the Object Manager tries to execute these waiting operation requests. Each queue element has the following structure:

```
struct wqueue
{
  request *Req;// pointer to a local request
  char Request[MAX_REQ]; // a remote request
  struct wqueue *back;
  struct wqueue *next;
};
```

Classical transactions are serialized by analysis of Read/Write and Write/Write conflicts between concurrency transactions. Techniques which allow more concurrency through additional interleaving than are permitted using these conflicts have been proposed [2] which employ type-specific properties of objects to recognize when certain operations such as concurrent write operations need not conflict.

The most important Object Manager that we developed, the File Manager uses a B++ tree-like structure to support indexed sequential access. The structure of the File Manager is shown below:

```
class FileManager: public VirtualDataServer, IndexedB
{
public:
  FileManager(char*, WIN*, unsigned, CHANNEL*, CHANNEL*, CHAN-
NEL*, CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*); //construc-
tor
  int Read(char,char*);
  int Write(char,char*);
  int Delete(char,char*);
  int Modify(char,char*);
  void Recovery(char*);
private:
  char Name[MAXNAME];
  int IsConflict(char, char, char*,int, char, char, char*, int); // verify the
concurrency
  int Switch(char,char,char);
  void MsgToNameSrv(void);//send initial information
  void file_open(char*);
  void file_close();
};
```

For this type of objects we defined four types of operations: Read, Write, Delete and Modify. We used for concurrency control a pessimistic scheme. Conflicts between concurrent transactions are identified during transactions' execution and resolved by imposing a delay (time–out) on operations until the transaction is aborted. Conflict-based concurrency control is based on predefined conflicts between pairs of operations. The purpose of the IsConflict function is to detect conflicts between the operation whose execution is requested and other operations already executed. Therefore, IsConflict function receives two pairs of arguments, one for the requested operation and the other for an operation already executed. Each pair of arguments contents: the type of the operation, the object identifier, pointer to the key of that record, and the transaction identifier.

Identification of the conflict types between the operations of concurrently active transactions is accomplished by a set of dependencies. These dependencies can be obtained directly from the abstract data type's specifications. In File Manager, the transactions are serialized by analysis of the Read/Write and Write/Write conflicts between concurrent transactions.

The transaction Server coordinates the initiation and termination of transactions. Therefore, Transaction Server must know always the state of every transaction active at his node. The Transaction Server structure is shown below:

```
class TransactionServer: public Task
{
public:
  TransactionServer(char*,WIN*,unsigned, CHANNEL*,CHANNEL*,
CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*); //
constructor
  virtual void TaskMain();
protected:
  CHANNEL *channel_a_ts, *channel_ts_a;
  CHANNEL *channel_ts_rs, *channel_rs_ts;
  CHANNEL *channel_ts_cs,*channel_cs_ts;
  CHANNEL *channel_ts_om, *channel_om_ts;
private:
  ID_tab IdTab[NRIDTR];// array of transaction identifiers
  ID_S_tab IdSTab[NRIDTR*NRSRV];//node's children; if the transaction
was initiated in many nodes these are necessary in two phase commit protocol
  char mesTran[CHANN];//messges received
  int i_IdTab, i_IdSTab;//number of transactions, number of node's children
  int trIdLimit;// the limit of the transaction identifiers allocated for this
node
  int SearchId(int);//search an identifier in IdTab
  void DeleteId(int) ;//delete an identifier in IdSTab
  void InsertIdS(char*);//insert an element in IdStab;
  long tran;//global semaphores for mutual exclusion
  WORD nr;// active workstation number in DOSTP
};
```

For every active transaction, the Transaction Server reserves an identifier. This identifier is used in DOSTP system by all the following requests from that transaction. To commit or abort a transaction the Transaction Server uses a variant of tree-structured two phase commit protocol; each node is the coordinator for the nodes that are its children. The Transaction Server keeps the information about a node's relation below in the tree in the IdSTab array. Each element of this array has the structure shown below:

```
struct ID_S_tab
{
  int id;// transaction identifier//    char ObjectType;//object type;identifies
the Object Manager's name
  int ObjectManagerConnection;// the connection number of the node in wich
ObjectManager resides
};
```

The Communication Server is the process that provides location transparent communication between applications and Object Managers; its structure is shown below:

```
class CommunicationServer: public Task
{
  friend void far loadds RecESR();
public:
  CommuniacationServer(char*,WIN*,unsigned, CHANNEL*,
CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*,
CHANNEL*); // constructor
  ˜CommunicationServer();//destructor
  request *req[MAXREQ]; //array of local requests' addresses
  int Find(char, char);//find an Object Manager
  virtual void TaskMain();
protected:
  CHANNEL *channel_cs_ts, *channel_ts_cs;
  CHANNEL *channel_cs_a, *channel_a_cs;
  CHANNEL *channel_cs_om, *channel_om_cs;
  CHANNEL *channel_ts_om, *channel_om_ts;
private:
  struct com S;//IPX connection's information;
  int TransId[NRIDTR];//number of transactions;
  cgar *MessageQueue[MAXREQ];//array of messages received by IPX;
  long semE;//NetWare semaphore for mutual exclusion in emision;
  WORD nrpE; number of DOSTP nodes
  SourceAddress TabSourceAddress[MAXREQ];keeps the parent node's ad-
dress
  int indTransId, indQueue, reqId, errTimeOut, errPreTimeOut; // indexes
  void Init(); //IPX initialization;
  int SearchId(int);// find a transaction identifier in the TransId table
  void InsertId(int);//insert a transaction identifier in the TransId table
  void GetMessQueue(char *);//get a waiting message
  void PutReqTimeOut(struct request*); //put a request in the time out
table;
  void GetReqTimeOut(int,char*);//get a request from the time out table;
  void PutPrepareTimeOut(char*);//put a PrepareToCommit
message in the time out table
  void GetPrepareTimeOut(char*);// get a PrepareToCommit
message in the time out table
  void TimeOutCheck();//test time out;
  void SourceAddr(int, int, char*); //record the parent node's address
  int Receive();//initiate a IPX reception
  void Send();//initiate a IPX emision
  void BuildPacket(request*);//build a packet
  void RecProcessing(char*);//analyse a remote packet
  void EmiProcessing(char*);//prepare a remote packet
};
```

The mechanism within the Communication Server supports the IPX protocol. The IPX protocol is a datagram delivery service. Every event, such as receiving or transmitting a packet is controlled by a structure known as an Event Control Block.

The Communication Server's Task Main function performs the following actions:

1. At the beginning of its execution, it initializes the communication channels (open the sockets, initializes the Event Control Blocks) and calls the Receive function in order to activate an asynchronous reception of a message. Whenever the Communication Server receives a packet from another node, the RecESR function is called. This function is declared as an Event Service Routine (ESR). ESR's are called by IPX with interrupts disabled. Upon completion of the ESR, IPX re–enables interrupts and returns control to the process interrupted by the event. Meanwhile, IPX listens for and attempts to receive a packet. The RecESR function places the incoming message in the MessageQueue.

2.The main loop of Task Main consists of receiving and sending messages. The Communication Server may receive messages from the application (in channel_a_sc), Object Manager (in channel_om_sc) and Transaction Server (in channel_ts_sc) and may send messages to Object Manager (in channel_sc_om) and Transaction Server (channel_sc_st). After initialization, the Task Main tries to find a remote message in MessageQueue; if the MessageQueue is not empty, then the first message is read and discarded; if the message type is valid, then the RecProcessing is called with this message as argument.

The Communication Server is designed to maintain an acceptable level of reliability in the transfer of information. Periodically, in order to check if, after sending a request a time out error occurs until the response is received, the Communication Server calls the TimeOutCheck function.

The Communication Server may receive and send remotely the following message types: R-operation request received from a remote application and forwarded by the Communication Servers, P-Prepared to commit message received from the Transaction Server, C-Commit message received from the Transaction Server, A-answer message sent to a remote application and forwarded by the Communication Servers, K-PrepareToCommitAck message, sent to the Transaction Server, the coordinator of the transaction.

The Communication Server may send locally the following message types:

1–confirmation request message sent to Transaction Server in order to confirm that a transaction identifier exists, 2–join message sent to a Transaction Server in order to record the new child node involved in transaction, 3,4,5–PrepareToCommit, Commit, Abort messages sent to the Local Object Manager and 6–PrepareToCommitAck message sent to Transaction Server, the transaction coordinator node. The Communication Server may receive locally the following message types: 1–request message received from an application, 2–transaction identifier's confirmation message, received from the Transaction Server, 3,4,5–PrepareToCommit, Commit, Abort messages received from Transaction Server, the coordinator of the transaction, in order to be sent remotely and 6–PrepareToCommitAck message.

The Communication Server uses a naming service (The Name Service) that provide permanent names for objects and Objects Managers. The structure of a Name Service is shown below:

```
class Name Service: public Task
{
public:
  Name Service (char*, WIN*, unsigned, CHANNEL*, CHANNEL*, CHAN-
NEL*, CHANNEL*, CHANNEL*,
CHANNEL*, CHANNEL*);
  Ñame Service();
  virtual void Task Main();
protected:
  CHANNEL *channel_om_ns, *channel_cs_ns, *channel_ns_cs;
private:
  struct infObjectM[MAXOBJ];// information about Object Managers
  long semE, semR, semER;// global semaphores
  Get Address(void);
  void Init(void);//IPX initiation
  void Reception(void);// initiate an IPX reception
  void Emision(void);// initiate an IPX emision
  void Send(void);//broadcast information about the new serve
  void Receive(void);//records a new server
};
```

An application task has the folowing structure:

```
class Application: public Task
{
public:
  Application (char*, WIN*, unsigned, CHANNEL*, CHANNEL*, CHAN-
NEL*, CHANNEL*, CHANNEL*, CHANNEL*, CHANNEL*);
  Ãpplication();
  virtual void TaskMain();
protected:
  CHANNEL *channel_a_sc, *channel_sc_a;
  CHANNEL *channel_a_ts;
  request *Req[MAXREQ];//array of pointers to requests
  int indReq, nrReq;//index of the first free slot in the request table, number
of requests
  int BeginTransaction();
  int EndTransaction(int);
  void AbortTransaction(int);
  int Synchronize;
  int BuildRequest(int, char*, char*);
  void SendRequest();//send the request
  void MsgDispatch();
};
```

6. Conclusions

The present paper describes the design and implementation of a distributed object–oriented system based on transaction processing, called DOSTP system. The implementation was done on a local network of DEC Stations 3000 under OSF/1 vers 1.32 and Zenith PC under Linux 1.2.8 and on a NOVELL network under NetWare 3.11. Basic transport-level communication is performed by TCP/IP, UDP/IP via an Ethernet and by IPX, respectively.

The major feature of this approach is the location independent object invocation; this is very useful for applications development. A programmer must never know where an Object Manager resides. Applications can be developed and debugging in a single node and then moved to the network.

DOSTP system provides a collection of functions integrated with a mechanism for handling concurrency and recovery.

We intend to implement the same basic components in a heterogeneous network based on UNIX and Windows NT operating system.

A conclusion from this work is that object-oriented design and transaction–based programming techniques can provide significant benefits in distributed systems.

REFERENCES

1. G. Andrews: *Paradigms for Process Interaction in Distributed Programs.* ACM Comput. Surv, vol 23, no 1, pp 49-90, Mar. 91
2. M.S. Atkins and M. Y. Coady: *Adaptable Concurrency Control for Atomic Data Types.* ACM Trans. Comput. Syst. , vol 10, pp 190-225, Aug. 9
3. H. Bal: *Programming Distributed Systems.* Silicon Press 92
4. N. S. Barghouti and G. Kaiser: *Concurrency Control in Advanced Database Applications.* ACM Comput. Surv, vol 25, pp 269-321, Sept. 1991
5. H. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in database Systems.* Addison Wesley, 87
6. L. F. Cabrera, J. A. McPherson, P. M. Schwarz and J. C. Wyllie: *Implementing Atomicity in Two Systems: Techniques, Tradeoffs and Experience.* IEEE Trans. Software Eng., vol SE-19, pp 950-961, Oct. 1993
7. R. S. Chin, T. Chanson: *Distributed Object-Based Programming Systems.* ACM Comput. Surv., vol 23, pp 91-125, Mar. 91
8. V. Grassi, L. Donatiello and S. Tucci: *On the Optimal Checkpointing of Critical Tasks and Transaction.* IEEE Trans. Software Eng., SE-18, no 6, pp 72-77, Jan. 1992
9. E. Delattre, J. M. Geib: *OMPHALE, an active objects-based distributed operating system.* in Proc Working Conference on Decentralized Systems, pp 327-340, Lyon 1989
10. P. Fortier: *Design of Distributed Operating Systems.* Mc Graw-Hill, 1988
11. B. W. Lampson: *Atomic Transactions in Distributed Systems-Architecture and Implementation.* Berlin, Germany: Springer-Verlag, pp 246-264, 1981
12. C. Musatescu: *Distributed Transactions in DISTRIB.* in Proc. 10th International Conference on Control System and Computer Science, vol 2, pp 287-291, Bucuresti, 24-26 May 95
13. C. Musatescu: *DISTRIB-A Distributed Object Oriented System Based on Transactions Processing.* in Proc. 5th International Conference on Control System and Computer Science, vol 2, pp 157-163, Iasi, 26-27 Oct 95
14. C. Musatescu, D. Burdescu: *A Novell Implementation of a Distributed Object Oriented System Based on Transactions Processing.* in Proc. 8th Symposion on Control System and Computer Science-SINTES, Craiova, Romania, Jun 96
15. C. Musatescu, D. Burdescu: *The Client-Server Model in a Distributed Object Oriented System Based on Transactions Processing.* in Proc. 8th Symposion on Control System and Computer Science-SINTES, Craiova, Romania, Jun 96
16. C. Musatescu: *Design and Implement a Multiuser Toolkit.* in Proc. 5th International Conference on Technical Informatics, vol 4, pp 41-48, Timisoara, Romania, 16-19 Nov 94
17. C. Musatescu: *Operating Systems Services for Database Management.* in Proc. of the 5th International Conference on Applied and Theoretical Electrotechniques, pp 310-315, Craiova, 18-20 Nov 93
18. G. Nutt: *Centralized and Distributed Operating Systems.* Prentice Hall 92

19. A. Z. Spector: *Support for distributed transactions in TABS prototype.* IEEE Trans. Software Eng., vol SE-11, no 6, pp 520-530, June 85
20. A. Z. Spector, P. M. Schwarz: *Transaction: A construct for Reliable Distributed Computing.* Technical Report, Carnegie Mellon Univ., Apr. 83
21. K. Ramamritham, B. R. Badrinath: *Synchronizing Transactions on Objects.* IEEE Trans. Comp., vol 37, no 5, pp 541-549, May 1988
22. P. Taylor, V. Cahill, M. Mock: *Combining Object-oriented Systems and Open Transaction Processing.* The Computer Journal, 37, pp.487-498, June 1994
23. A.Tanenbaum: *Computer Networks.* Prentice-Hall, 1988