DEVELOPING DISTRIBUTED APPLICATIONS WITH DISTRIBUTED HETEROGENEOUS DATABASES

by

Eric Richard Dixon

Project submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science & Applications

APPROVED:

C. J. Egyhazy, Chairman

W. B. Frakes

J. E. Kohl

May, 1993

Falls Church, Virginia

Introduction

This report identifies how Tuxedo fits into the scheme of distributed database processing. Tuxedo is an on-line transaction processing (OLTP) system. The reason that Tuxedo is being studied is that it is the oldest and widely used transaction processing system on UNIX. The fact that it is the oldest and most widely used OLTP system for UNIX means that it is established, less buggy, and has the most tools available to extend its capabilities. The disadvantage of Tuxedo is that newer UNIX OLTP systems are often based on more advanced technology. For this reason, other OLTPs will be examined in comparison to Tuxedo later in the report to examine additional capabilities they offer.

The report first establishes a framework for the study of Tuxedo by analyzing Tuxedo in regard to the Distributed Transaction Processing (DTP) model. The DTP model (developed by the X/Open organization) specifies how OLTPs should be structured in order to support all of the standards related to the model. The DTP model is the architecture used by modern UNIX OLTP systems. The DTP model has three main components: application programs (APs), transaction managers (TMs), and resource managers (RMs).

X/Open either has standardized or is standardizing communications between all the pieces of the DTP model (APs, TMs, and RMs) both horizontally and vertically. Vertical communications involves communications between different pieces of the model (e.g. between a TM and an RM). Horizontal communications involves communications between different instances of pieces of the model at the same level (e.g. between two TMs). Tuxedo conforms to all the current X/Open standards related to the DTP model and will support additional standards as they are adopted.

The report then uses that framework to make a study of the Tuxedo on-line transaction processing system in order to illustrate how Tuxedo fits into the scheme of distributed database processing. The method for the study is by comparing Tuxedo and the DTP model it is based on to C.J. Date's model of distributed database management systems. The basis for comparison is the result of studying literature about Tuxedo, transaction processing, on-line transaction processing, and the DTP model and comparing the results of that study to Date's model of distributed transaction processing systems. The primary topic of Date's model studied is his five problems of distributed database processing systems: query processing, update propagation, concurrency control, commit protocol, and catalog management.

The first three sections of the Report discuss the DTP model, concepts and terms related to the model, and standards associated with the model. The subject of the first section is transaction processing. Topics discussed in the section include transactions, transaction processors, global transactions, and on-line transaction processors. The second section is about the DTP model. Subjects in this section include a discussion of the X/Open Organization, the DTP Model, and the OSI TP standard. The third section is about the standard which specifies the interface between TMs and RMs, the XA specification. Topics of this section include responsibilities of the XA specification, threads of control, the "C" language API for the XA interface, and problems with implementations of the XA specification.

The rest of the report is about Tuxedo. The fourth section of the report contains an overview of Tuxedo, a discussion of the enhanced client/server model (supported by Tuxedo), a discussion of Enterprise Transaction Processing (also supported by Tuxedo), an example of an application suited for Tuxedo, and a discussion of future capabilities which will be supported by Tuxedo.

The fifth section discusses five distributed database problems discussed by C. J. Date and how those problems are addressed in Tuxedo systems. The problems are query processing, update propagation, concurrency control, commit protocols, and catalog management. The sixth and final section discusses Tuxedo in comparison to other modern OLTPs, including: Transarc's Encina, Top End, and CICS. This section discusses the capabilities of the different OLTPs in comparison to Tuxedo. The section does not give a detailed technical discussion of those products.

Table of Contents

List of Figures

Nomenclature

2PC	Two-Phase Commit
AP	Application Program
API	Application Program Interface
ATM	Automated Teller Machine
CAE	Common Applications Environment
CICS	Customer Information and Control System
COBOL	COmmon Business Oriented Language
DBMS	DataBase Management System
DDL	Data Definition Language
DLL	Dynamic Link Library
DML	Data Manipulation Language
DOS	Disk Operating System
DTP model	The Distributed Transaction Processing model
FTAM	File Transfer, Access, and Management
GEIS	General Electric Information Services
HP	Hewlett-Packard
IBM	International Business Machines
IPC	InterProcess Communications
ISAM	Indexed Sequential Access Memory
ITI	Independence Technologies, Incorporated
LAN	Local Area Network
OLTP	On-Line Transaction Processor
OS/2	Operating System/2
OSI	Open Systems Interconnect
PC	Personal Computer
PIN	Personal Identification Number
POS	Point Of Sale
RDBMS	Relational DataBase Management System
RM	Resource Manager
RPC	Remote Procedure Call
TM	Transaction Manager or Transaction Monitor, the terms are synonymous
TP	Transaction Processor
TRS	American Express's Travel Related Services
USL	UNIX System Laboratories
WS	Work Station
XA	X/Open's standard protocol for communications between TMs and RMs
XID	Transaction Branch Identifier

Section I. Transaction Processing

Tuxedo is an on-line transaction processor. This section discusses transactions and transaction processing. The section begins with a discussion of transactions. The next subsection discusses what transaction processors are and the evolution of transaction processor capabilities from the original, simple transaction processors which performed transactions on single databases to transaction processors which are capable of supporting global transactions across distributed, heterogeneous database managers.

Following the discussion of transaction processors is a discussion of global transactions. The discussion of global transactions defines global transactions, transaction managers (who coordinate global transactions), transaction branches (pieces of global transactions), and the two-phase commit protocol which is used to commit and roll back global transactions. The last topic of discussion in this section is about on-line transaction processors.

1.1 Transactions

A transaction is a complete unit of work. For example, if you go to an Automated Teller Machine (ATM) at your bank, insert your card, type in your Personal Identification Number (PIN) and withdraw cash, that is a single transaction. All work to accomplish the withdrawal is included in the transaction. For example, the ATM transaction includes typing in the amount of your withdrawal, verifying that you have enough cash to make the withdraw, verifying that you do not exceed the daily ATM withdrawal limit, verifying that the machine has enough cash to give you the amount of cash that you asked for, and actually giving you the cash.

If any step in the transaction fails or if the user changes their mind before the transaction is complete, the transaction must be "rolled back," or canceled. For example, if you ask for more cash than you have in your account, or if you press the cancel key on the ATM machine before the transaction is complete then the transaction must be rolled back. When a transaction is rolled back, the data (in this case the account balances) are as if no transaction had ever been performed. How would you feel if you asked for a transfer from one account to another and after withdrawing the money from one account but before depositing it into another account, the transaction failed and your money was gone. Transaction processing must not allow that to happen.

Once transactions are determined to be complete, then the work done by the transaction is "committed." Once the work is committed, the work becomes permanent. If you do an account balance on two accounts after transferring money from one to the other, either they should both show the affects of the transfer or neither should show the affects of the transfer. Also, if there is a failure in the system and the transfer has been committed, sufficient recovery mechanisms need to be in place in the system so that neither the transfer nor any part of the transfer is lost.

One more important concept of transactions is that outside processes should not see partial results of the processing of the transaction. For example, if your monthly statement is being calculated while you are withdrawing cash, but before you complete the transaction, your statement should not reflect any of the partial results of the withdrawal. The monthly statement processing may be blocked while the transaction is completing or it may simply receive the account balances before the transfer, but it will not see any of the partial results while the transfer is performed.

1.2 Transaction Processors

Avraham Leff and Calton Pu define transaction processing systems as database systems which support the concept of transactions.[1] Two key requirements of transaction processors are recovery atomicity and concurrency atomicity. In recovery atomicity, once a transaction begins, it will either complete successfully or make no changes to the database. In concurrency atomicity, the interleaved executions of one process performing a transaction on a database have no affects on the processing of another. These atomicity rules "are intuitively appealing because they allow users to think of a complex database application program as an atomic, fail-safe modification of the database."

Leff and Pu discuss the development of transaction processing systems. The original and simplest transaction processing systems "run on one machine and support only one view of data." However, different users of a system have different views of the "external world". The next evolutionary step of transaction processors, therefore, was to allow multiple views of the data in the databases in order to support the different views different users have of the real world, represented by the data.

Transaction processing systems then began to expand onto multiple machines with homogeneous database managers. Multiple machines leads to "better response time, reliability, and availability." The price of those benefits was increased overhead to coordinate "atomicity, serializability, and data integrity" across the machines. The homogeneous database managers were used to minimize the already greatly increased complexity. Eventually, database management systems evolved to a level where it was no longer necessary to use homogeneous database managers, and heterogeneous database managers were supported.

1.3 Global Transactions

Transaction processors can perform transactions which span multiple databases on multiple machines. The databases could even be physically located around the world. It is clearly more complicated to commit and roll back transactions which span multiple databases than to commit and roll back transactions which only span a single database. When transactions span multiple databases, multiple resource managers must be involved in committing and rolling back the transactions. We will now discuss global transactions (which span multiple databases), transaction managers (which manage global transactions), transaction branches (a piece of a global transaction related to a single database), and the two-phase commit protocol (used to commit or roll back global transactions). The definitions of the terms in this subsection are based on the definitions in X/Open's XA Specification document.[2]

Transactions which span multiple databases are called global transactions. For example, General Electric (GE) has an Aircraft Engine business and a Medical Systems business. In order to improve performance and make maintenance of the data easier, the data related to each business is kept in databases in physical locality of the business. If the databases were kept on some sort of connected network, corporate GE could perform global transactions spanning both databases to determine information such as how much money GE is making this year and the average salary of GE employees. Every database manager involved in a global transaction must support both the concept of transactions and global transactions.

Global transactions are atomic pieces of work, just like transactions. This implies that if a global transaction fails, the effects of the transaction on each site must be rolled back. In other words, if work which is part of a transaction on any site fails, work which is part of the transaction on that and all other

sites which are part of the transaction must be rolled back. Also, if a global transaction is committed and a failure occurs, no work which was part of the global transaction should be lost.

In the DTP model, global transactions are managed by a process called the Transaction Manager (TM). The TM is responsible for coordinating the transaction among the multiple sites involved in the transaction. Through the appropriate interface, the application tells the TM about the transaction which is to be performed (we will discuss this more extensively in the Distributed Transaction Processing section). The TM tells the database or resource managers (RMs) what work they are each to do, collects information as to whether each piece succeeded or failed, and coordinates committing or rolling back the transaction, and coordinates global recovery when failures occur.

A transaction branch is a piece of a global transaction related to a single database. When a global transaction involves a database operation, the TM tells the RM what needs to be done, and a transaction branch is created. A single branch relates to a single unit of work. A global transaction involves multiple transaction branches across multiple databases. There can be single or multiple transaction branches in each database involved in the global transaction. Each branch is assigned a unique identifier (called an XID) by the TM for reference by the TM and the RM involved in processing the branch.

The protocol used by the TM to commit and roll back global transactions is called the Two-Phase Commit (2PC) protocol. The 2PC protocol is defined in the OSI TP standard, which will be discussed in the Distributed Transaction Processing section. In the first phase of the protocol, the TM sends a "prepare to commit" message for each transaction branch. Each RM responds whether it is capable of committing the branch (i.e. whether that part of the transaction has succeeded or not). If a transaction branch is successfully completed, the related RM keeps the information necessary to apply the work to the database and responds that the branch completed successfully. If the branch is not successfully completed, the RM rolls back its part of the work and responds that the transaction branch was not successfully completed.

In the second phase of the 2PC protocol, the TM waits for the responses from all the transaction branches. If they all complete successfully, the transaction manager sends a really commit message for each transaction branch. If any of the branches do not successfully complete, the TM tells all the RMs to roll back their transaction branches. Once the transaction is complete, either all or none of the changes are applied to all the databases and the application is told whether the transaction was successfully completed or not.

1.4 On-Line Transaction Processors

Transaction processors fall into two basic categories: Batch Transaction Processing (Batch TP) and On-Line Transaction Processing (OLTP).[3] In batch processing, requests are queued in a block and run one after the other. When one request completes, the next begins. An example of batch processing would be a back office report that is run at night

In on-line transaction processing, transactions are run in real time. For example, a credit card authorization service would run as a real time point of sale (POS) application. When you are shopping with your credit card, you would not want to give the clerk your credit card number and come back the next day to see if your purchase was approved, you want your number verified immediately through an on-line transaction against the credit card company's database so that you can take your new golf clubs home with you. However, on-line transaction processors are capable of performing batch processing. Batch processing just means that one transaction follows another. Today, there is little distinction between batch and on-line transaction processing systems.

Gordon McLachlan, in an article in HP Professional magazine, says that On-Line Transaction Processors have been around for 20 years.[4] McLachlan says there have been only a few "proprietary, mainframe-based approaches" to on-line transaction processing, and IBM has historically dominated that market. Today, OLTPs are adopting client-server architectures and moving into distributed computing. These trends have lead to a "shift toward open, networked platforms." UNIX has increased in popularity and may be the "platform of choice" for OLTPs because of its good price/performance ratio. UNIX has also benefited from its "improved security, performance, usability, and networkability."

Modern OLTP architectures are usually based on X/Open's Distributed Transaction Processing (DTP) model, which will be discussed extensively in the next section. The DTP model includes a transaction manager, which "translates application requests into a form that the data resource manager can understand, sequences concurrent application requests, and coordinates the distributed two-phase commit process." Transaction managers based on the DTP model use the XA standard to communicate with resource managers. The XA standard will be also discussed in depth in a later section of the report. Some transaction managers use their own proprietary language in order to communicate with the resource managers. For example, IBM has historically used proprietary protocols for communications. Resource managers can be in the form of database management systems file system managers which have been adapted to use the appropriate protocol.

While the XA Specification defines the interface between TMs and RMs, there is currently no generally accepted standard for communication between the application and TM. Standards are being researched and developed in this area, but we do not yet have a generally accepted standard. Therefore, current OLTP systems have their own APIs for application programs to supply the "syntax and semantics for programmers to group database operations into transactions." Transaction managers generally require some sort of "begin transaction" command, allow database operations, and then require the program to indicate the end of the transaction with a "commit" (apply the changes to the database) or "rollback" (do not apply the changes to the database) command.

There are currently three major OLTPs for UNIX. UNIX System Laboratory's (USL's) Tuxedo has the largest market share of any of the OLTP marked and is two years of development ahead of any other product. USL was recently sold to Novell, who now owns Tuxedo. NCR was developing a product called Top End when they merged with AT&T, and Top End is now the combined companies "strategic transaction manager." Until recently, Top End only ran on NCR 3000 computers. It has recently been ported to HP computers and is in the process of being ported to other platforms.

The third UNIX OLTP is Transarc's Encina. Encina uses the most advanced technology; however, Encina is not yet commercially available. Users will also not be able to use personal computers as workstations when they are running Encina. IBM has also announced plans to port CICS to UNIX running on top of Transarc's Encina. CICS is currently one or more years away from becoming commercial on UNIX. An overview of these OLTPs and how they compare to Tuxedo will be given later in Section VI.

Section II. The Distributed Transaction Processing Model

Current on-line transaction processing system vendors almost all design their transaction processors according to X/Open's (a independent standards organization) Distributed Transaction Processing (DTP) Model. Tuxedo is also designed according to the model. This section begins with a discussion of the X/Open organization. Next, the pieces of the DTP model are defined and the DTP model is described. The major pieces of the DTP model are the Application Process (AP), Transaction Monitor (TM), and Resource Managers (RMs).

The final topic of the section is a discussion of the OSI TP model. The OSI TP model specifies the definition of transactions referenced by the DTP Model. The OSI TP model also describes the communications protocol used between the processes of the DTP model. The DTP model specifies interfaces between parts of the DTP model and assumes that the OSI TP model will be used as the base of communications. The subsection will not go into an in depth discussion of the OSI TP model, but will discusses the model in terms of its relevance to the DTP model.

2.1 The X/Open Organization

X/Open discusses the nature of their organization in their XA Specification document. X/Open is an independent organization of information suppliers, users, and software development companies. X/Open attempts to evaluate and integrate "existing and emerging standards" to create an environment it calls the Common Applications Environment (CAE). The CAE is an open systems environment with comprehensive standards covering all levels of systems above the hardware level. The goal of the environment is to provide "portability and interoperability of applications" and to allow "users to move between systems with a minimum of retraining."

X/Open selects and adopts existing standards, wherever possible. Where there are no existing standards, X/Open works with other standard development organizations "to assist in the creation of formal standards covering needed functions." X/Open's goal is communication and standardization. Therefore, X/Open freely provides its work to standards development organizations. X/Open also "has a commitment to align its definitions with formal approved standards."

X/Open has two levels of specifications. The CAE specifications are the complete X/Open approved specifications. Developers can be sure that when they follow CAE specifications, the products of the organizations participating in the standard will either currently be consistent with the standard or will be consistent with the standard in a future release. The users also know that they will have a variety of systems which will be capable of hosting their products.

The second level of specifications are preliminary specifications. Preliminary specifications are generally "addressing an emerging area of technology," and thus have not had the necessary base of implementations or prototypes to indicate general acceptance of the standard. The standards are not unpublished preliminary standards. They have in fact been as tested and documented as any CAE specification. The difference is that they are not yet in wide enough use to be adopted as an industry standard.

X/Open also publishes "Snapshots," which are formal documents which provide information to X/Open's community presenting its "current direction and thinking" for the purpose of "soliciting feedback and comment." Snapshots do not provide final standards, but interim results. "X/Open is a consensus organization" and therefore its mission is accomplished when it is publishing standards desired by the majority of its standard followers.

2.2 The DTP Model

X/Open's XA Specification document defines their Distributed Transaction Processing (DTP) model. The DTP model envisions three parts: an application program (AP), a resource manager (RM), and a transaction manager (TM). We will now discuss each part in greater detail.

The application program (AP) is an instance of a process, written by application programmers, to "define transactions and access resources within transaction boundaries." In the case of the ATM withdrawal, the application program is the process that determines that when the user presses the withdraw button, the user must be asked how much to withdraw, a query must be done to determine that the user has enough money in his account to withdraw the requested amount of money, and that no withdrawal limits are violated. The application program then also says what databases must be updated to record the withdrawal and directs the ATM machine to give the user the appropriate amount of cash. If the user presses the cancel button, the AP may initiate the roll back command. There can be many different APs in a DTP system and there can be multiple instances of a particular AP.

The transaction manager (TM) "assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery." The AP communicates with the TM, who coordinates processing of the transaction. Details of the processing of the transaction are controlled by the TM and the results are given to the AP. This makes application programming easier since the TM is vendor supplied and the application developer does not need to program the details of coordinating global transactions. TMs were discussed more fully in the global transaction subsection of the Transaction Processing section.

The other major piece of the DTP model is the resource manager (RM). The RM defines how database and file systems "provide access to shared resources." In our discussion of transaction processing so far, we have interchanged database management systems and resource managers. In general, resource managers are processes which manage access to shared resources. Resource managers can be DBMSs, file system managers (e.g. an ISAM file access system), or a printer. For a RM to participate in a global transaction in the DTP model, it must be capable of supporting the protocols (e.g. OSI TP and XA Specification) which are part of the DTP model. Just like APs, there can be multiple types of RMs in the system and multiple instances of the same type of RM.

Next is a figure from the XA Specification document showing the X/Open DTP Model:

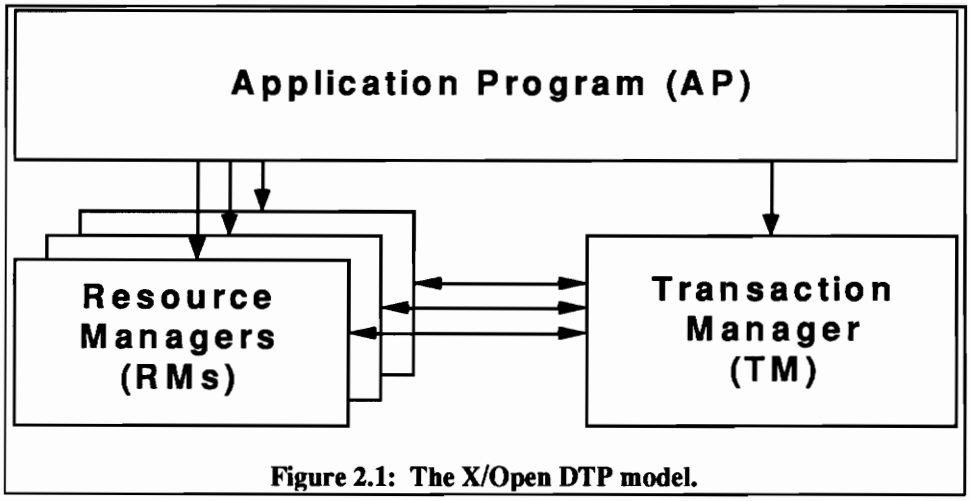

Figure 2.1: The X/Open DTP model.

The lines show the exchange of control information. All of the pieces could be on the same or different processors and located in the same or different processes. The lines from the AP to the RMs indicate that applications can make direct calls to the RMs without being in global transaction mode. If the transaction monitor is bypassed in this way, the application is responsible for managing the transaction with the RM. The AP can use single phase commit to control these local transactions. The X/Open recognizes SQL as the standard for the interface between an AP and a RM.

While standard committees are working on standards for the interface between the AP and the TM, there is currently no established standard recognized by X/Open. Therefore, different TM vendors use proprietary protocols for this purpose. The implication of this is that it is difficult to port code from one TM vendor to another and it is difficult to use different TMs to work together on the same transactions. It is X/Open's goal to eventually have all interfaces standardized for maximum portability and interchangeability of the pieces.

The XA Specification specifies the standard interface between the RMs and the TM in the DTP model. The XA Specification is not just an Application Program Interface (API), but a specification of the "system-level interface between DTP software components." The XA Specification will be discussed extensively in the next section. The XA Specification does not define the communications layer protocol. X/Open states that the OSI TP protocol is to be used for communications.

2.3 The OSI TP Model

This section gives more information about the Open Systems Interconnect TP (OSI TP) model. The section will not discuss the details of the model, but gives an overview of what the model was developed for and how it aids in transaction processing. The intent of the model is to define how transactions are handled in open systems processing.

Each AP, TM, and RM in the DTP model is comprised of one or more processes. Each piece is also supplied by a different source. TMs are supplied by TM vendors who specialize in TM functionality. Similarly RMs are supplied by vendors (e.g. DBMS vendors). AP programmers can purchase the completed TMs and RMs to manage databases and global transactions in their applications. The DTP

model assumes that the messages specified in its interface standards are delivered without discussion of the underlying delivery layer. The underlying layer is responsible for delivering interface messages reliably and without errors. The increase in modularity of the processes within the DTP system increases the efficiency and reliability, but also requires better interprocess communications (IPC).[5] X/Open specifies that the communications will be done using the OSI TP Model, which is expected to accomplish these goals.

Lee Mantelman, in an article in Data Communications magazine, describes the origins of OSI TP.[6] In the article, Lee Mantelman quotes John Neumann that OSI TP is "the first real OSI application protocol from the user's point of view." Neumann says that the X.400, X.500, and FTAM (File Transfer, Access, and Management) protocols are not "solving real application problems. OSI TP can be used by any application requiring interprocess or reliable communications."

Mantelman also says that, historically, IBM has dominated the OLTP market. IBM's uses LU6.2 to perform the capabilities that OSI TP performs. OSI TP is an attempt for vendors to compete with IBM for market share. LU6.2 is based on IBM's proprietary SNA protocol stack and anyone who implements a version of LU6.2 must pay IBM royalties. Also, some of the capabilities of OSI TP go beyond LU6.2. Despite this, a quarter of the standard creators of the OSI TP protocol worked for IBM. Helmut Wrubel, Siemens AG's system architect and one of the developers of OSI TP says that "IBM is interested in every country where it has premises." On the other hand, Clive Partridge, director of Data Connection Ltd, says IBM is involved in OSI TP "because they always hedge their bets."

Section III. The XA Specification

In previous sections, we discussed the DTP model. The DTP model references the XA Specification as the standard interface between transaction managers (TM) and resource managers (RMs). This interface is a very important part of the DTP model and will be discussed in this section. First, a discussion of the specification will be presented. We will then discuss threads of control. A description of associating threads of control with resource managers and how the two phase commit protocol is controlled using the "C" interface to the XA Specification will be shown. Finally, in real life nothing is perfect. Some problems which are encountered with the use of the XA Specification will be discussed. The descriptions of the XA Specification are in reference to X/Open's XA Specification document.

3.1 Responsibilities of the XA Specification

The XA specification includes the functionality necessary to implement the two-phase commit protocol for committing and rolling back global transactions. In phase I of the two-phase commit protocol, the TM tells the RMs to "prepare to commit" the transaction. If any of the RMs do not successfully complete this phase, then the TM instructs each RM to roll back the transaction. The AP can also instruct the TM to roll back the transaction. If the Global Transaction is rolled back, then none of the processing done in phase I should become permanent. The XA specification gives the interface necessary for the TM and RMs to perform two-phase commit.

The XA specification also allows certain optimizations in the protocol to occur automatically. For example, if an RM is asked to read data from a database without writing data to the database in a transaction branch, the RM can inform the TM in its response to the phase I "prepare to commit" message that no database updates are necessary and the TM will automatically drop the phase II messages to that RM. Another automatic optimization is that if the TM knows that only one RM is involved in a global transaction, it may instruct the RM to do a simple one phase commit on that database.

DTP systems must be "able to recover from a variety of failures. A storage device or medium, a communication path, a node, or a program could fail." When possible, RMs should correct the problem internally without affecting the global transaction. The next best case for recovery is that the commitment protocol is not compromised and a global rollback can be performed. When global recovery mechanisms are required, however, the TM is responsible for coordinating that recovery. The XA specification must describe an interface between the TM and RMs that supports the necessary recovery mechanisms.

3.2 Threads of Control

A thread of control, or thread, is the operating system process that is currently in control of the processor. A thread has an address space and a context. The context includes objects such as open files, semaphores, and shared resource locks. Threads are important for RMs in transaction processing. APs can call RMs to perform database operations and TMs can call RMs in transaction branches. Threads are how RMs keep track of the AP who is doing the work. In both cases, the RM is called from the same thread.

When an AP requests a TM perform a global transaction, the TM records and informs RMs that a global transaction is beginning. After control is returned to the AP, it makes the calls to the RM using the

standard interface for the RM. Both TM and AP messages are sent to the RM from the same thread. Threads can be tightly or loosely coupled. Tightly coupled threads share the same resources, and loosely coupled threads do not.

3.3 "C" Language API for the XA Interface

This section discusses the "C" language API for the XA Interface (the interface between the RMs and the TMs). We will discuss how threads are associated and disassociated from RMs. We will then discuss the routines the TM uses to call the RMs in order to create transaction branches and control the two-phase commit process. The convention for function names of the routines which are called by the TM to communicate with the RM is that the function name begins "xa_". The function names called by the RM to communicate with the TM begin "ax_".

The static routines used to associate and disassociate a thread from an RM are called xa_open( ) and xa_close( ). Other "xa_" calls can only be made between calls to xa_open( ) and xa_close( ). When the TM calls xa_open( ), the RM may perform initialization (like opening files) to prepare to process the transaction branches. The TM may also pass in parameters in the form of a string to xa_open( ). The call to xa_close( ) will also accept parameters in the form of a string.

Using xa_open( ) and xa_close( ) with RMs is called static registration because after xa_open( ) is called, transaction branches can be created freely by the thread. If a resource is not frequently used, it can use dynamic registration. In dynamic registration, the TM does not call xa_start( ) or xa_end( ). When the AP makes calls to the RM through the RM's native interface, the RM calls ax_reg( ), which asks the TM if the work is part of a global transaction. If the work is part of a global transaction, the TM returns the XID for the transaction branch to the RM. The RM calls ax_unreg( ) to notify the TM when the work is complete. The resource manager will declare that dynamic registration will be used by informing the TM after the TM calls xa_open( ).

Will now discuss how the two-phase commit protocol is performed. In phase 1 of the two-phase commit protocol, the function xa_start( ) is called by the TM to indicate creation of a new transaction branch. The AP then accesses the resource by its native interface. For example, if the resource is an RDBMS which supports SQL, the call to xa_start( ) would be followed by the AP performing SQL statements. Everything that is done before xa_end( ) is called is deemed to be part of the transaction branch. A thread can create multiple transaction branches using the same RM during the same transaction.

A special form of xa_end( ) allow transaction branches to be suspended. Suspended branches can then be resumed by a special form of xa_start( ). If the RM supports "association migration," then either the suspending thread or another thread could resume the transaction. If associative migration is not supported, then the thread which suspended the transaction must resume or end it later.

Once all the transaction branches are ended using xa_end( ), phase II of the two-phase commit process can begin. The TM calls xa_prepare( ) for each transaction branch and the RM responds whether it is capable of committing the transaction branch. If the RM responds yes, it must be capable of not losing the data related to the transaction branch even if a failure occurs. Finally, the TM calls xa_commit( ) or xa_rollback( ) for each transaction branch to commit or rollback the global transaction.

3.4 Problems with Implementations of the XA Standard

In theory, any XA compliant Transaction Monitor and Resource Manager can work together. McLachlan points out some problems with the way things work in the real world. One problem is that (as stated earlier) X/Open assumes that OSI TP is performing the communications between pieces. However, "the lack of explicit standards for a remote procedure call (RPC) mechanism, distributed naming service, security mechanisms, and the countless other minutiae needed to get dissimilar machines to communicate reliably is a real obstacle."

Another problem is that there is no standard protocol between the application process (AP) and the TM. X/Open is currently evaluating Tuxedo's TX protocol to determine if that should become the standard, but currently there is no standard. The lack of a standard in that area means that APs and TMs are "fused" together using proprietary APIs. The lack of a standard also means that while RMs may be interchangeable, TMs are not.

The other problem discussed by McLachlan is that old OLTP systems, like IBM mainframe systems do not conform to the standards. IBM is attempting to create a protocol and programming interface to integrate CICS into X/Open Standards using LU6.2 (a packet protocol commonly used by IBM computers). However, getting Transaction Managers on UNIX to work well with CICS will not be easily done since CICS is designed to work well on IBM mainframes while the other TMs are designed to run on UNIX boxes.

Section IV. Tuxedo

This section discusses Tuxedo itself. The first topic is an overview of Tuxedo. The next discussion is about the enhanced client/server model. Tuxedo applications are designed using the client/server model. Tuxedo allows clients to call servers synchronously or asynchronously. They call this capability the enhanced client/server model. The next topic in the section is how Tuxedo extends transaction processing from the UNIX platform to databases on proprietary mainframes and PCs. They call this capability Enterprise Transaction Processing (ETP).

After describing Tuxedo and its architecture, the next topic is a discussion of a major on-line transaction processing application which involves image processing. The project was done for American Express's Travel Related Services (TRS) division. The example is intended to show the power capable in on-line transaction processing applications. The final topic in the section is a discussion of future capabilities that Unix System Laboratories (USL) has announced that it plans to support in Tuxedo.

4.1 Tuxedo Overview

Tuxedo was originally developed by UNIX® System Laboratories, Incorporated (USL) in 1979 and has been commercially available since 1986.[7] Tuxedo was originally developed to support on-line transaction processing and allow database transactions to span multiple platforms running the UNIX System V operating system. In July, 1991, Tuxedo Release 4.2, extended transactions to databases on Personal Computers (PCs) and proprietary mainframe computers. This increased capability is referred to as Enterprise Transaction Processing (ETP) since the transactions are considered to span the databases of entire enterprises (or businesses). There are four main components to the Release 4.2 system: the Transaction Monitor (System /T), the workstation extension (System /WS), the host extension (System /Host), and the database management (System /D) system.

Tuxedo applications are written using the enhanced client/server model. Tuxedo systems are divided into clients and servers. Clients send messages, called service requests, to servers, who process the request and return a reply to the client. For example, a client could call a server, who does a database query, and returns the result of the query for the client. Servers can be located anywhere on a connected "network of loosely coupled computers running clients and servers within the framework of a single application." Furthermore, Tuxedo provides a high level interface (ATMI) which allows clients to make requests from servers without knowing where the servers are located. The enhanced client/server model will be discussed more fully in the next subsection.

Tuxedo's Distributed Transaction Processing (DTP) capability provides data integrity as messages are sent between sites throughout the network. Tuxedo also "coordinates distributed transactions to allow multi-site updates against heterogeneous databases on networked computers." Tuxedo supports the concept of global transactions. Using a two-phase commit protocol, the Transaction Monitor ensures global transactions will be committed and rolled back properly at each site. Tuxedo also provides recovery for site or network failures during global transactions and for global resource deadlocks. To enhance performance and throughput, Tuxedo uses algorithms to minimize disk writes and reduce network traffic. Tuxedo will also automatically reduce two-phase commit to one-phase commit when possible.

Tuxedo "supports X/Open's XA Interface standard for coordinating transactions among multiple database managers participating in an application." XA is part of X/Open's DTP model. Tuxedo's XA compliance allows any XA compliant database product to be used in Tuxedo applications. These databases can be replaced when required without changing any application code. Also, the two-phase commit protocol can be used by Tuxedo on heterogeneous database managers which could be running on multiple boxes using different hardware and running different operating systems.

Tuxedo conforms to the ANSI standard and is supported on most UNIX based platforms, including Hewlett-Packard (HP), Sun, DEC, DG, Pyramid, Tandem, Unisys, AT&T, Amdahl, Bull, Fujitsu, ICC, OKI, Olivetti, Sequent, Teradata, and Toshiba. Tuxedo is available for the following operating systems: UNIX System V Releases 3 and 4, Sun OS, AIX, UTS, VMS, Ultrix, HP UX, OS2 (/WS), MS-DOS (/WS), and MVS/CICS (/Host).[8]

USL owns and maintains the base Tuxedo code, but Tuxedo is ported to the various platforms and operating systems by vendors. The examples in this report have been developed using an Independence Technologies, Incorporated (ITI) port of Tuxedo to Hewlett-Packard platforms running the HP UX operating system. In addition to porting Tuxedo, ITI supplies additional products, including: iTran (eight software development tools), iView (an application controlling and monitoring product), iScreen (an object oriented GUI development tool), iNI (network interface software), and iDM (a data manager for message handling).[9]

4.2 The Enhanced Client/Server Model

Tuxedo servers provide services. Clients send messages to servers by calling services, by name, which are provided by the servers. Calls to services are called "service requests". A service request will automatically be routed by Tuxedo to a server which provides the requested service. For example, if a client wishes to call service A, then using a function call interface, the client makes a service request for service A. Suppose there are four servers in the system, two of which provide service A. The service request will be routed directly to one of the two servers which provides service A. The client does not know how many servers in the system provide service A, which server which provides service A will receive the service request, or where the server which receives the service request is located.

The next figure shows four clients and four servers which provide (collectively) services A, B, and C running on two boxes. Clients can be running on either box. All the clients and servers are connected to Tuxedo's Transaction Monitor (TM). If a client makes a service request for service A, the service request will be handled by server 1 or 2. A service request for service B could go to Server 1, 3, or 4. A service request for service C could go to server 2 or 4. Notice that no server on box 2 provides service A. That is not a problem for clients on box 2. Service requests for service A would be handled by a server on box 1 which provides service A without any special processing by the clients on box 2.

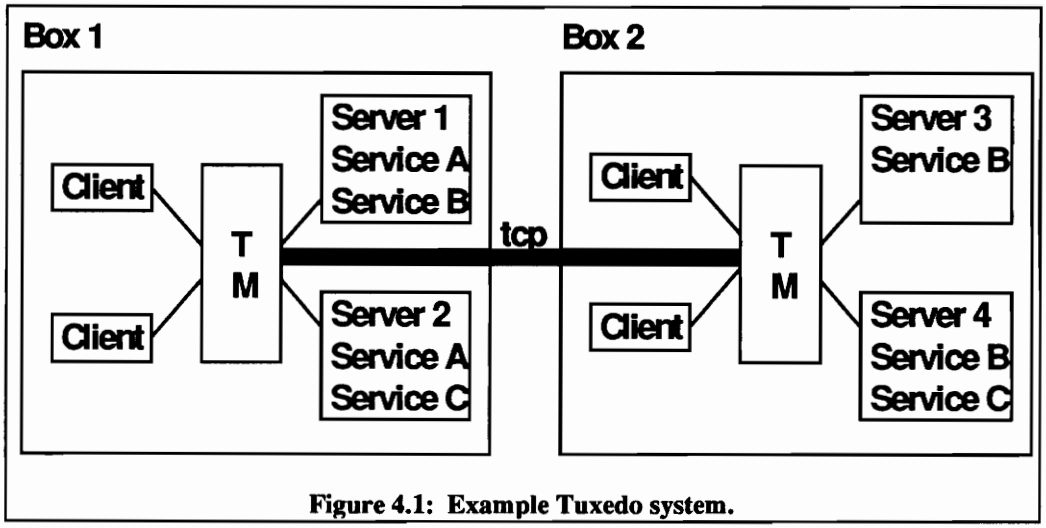

Figure 4.1: Example Tuxedo system.

Tuxedo's client/server model can be referred to a connectionless system. Connectionless systems are fundamentally different than the read/write paradigm in which two processes establish a connection (which may or may not provide message integrity) and exchange messages across the connection. When a server receives a message in a connection based system, it must parse the message and call a function which knows how to process the request. In Tuxedo, the client makes a service request which directly invokes the function in the server which knows how to provide the service, thereby skipping the parsing and decision making steps.

The connectionless system also has the advantage that since no connections need to be established, the overhead of establishing and maintaining connections in both the client and server is eliminated. When a client wishes to call a service, it does not need to establish a connection to the appropriate server and the server does not need to maintain an array of connections from its current clients. In Tuxedo, when a server completes processing a service request, the server gives the reply to Tuxedo who returns the reply to the client. In a connection based system, the server would need to remember which connection the request came across and return the reply on that connection.

The following diagram shows the layout of another Tuxedo system with database processing. In this example, we are not using a distributed databases with two-phase commit, but single databases. We will explore distributed databases in other sections. Services A and B access database α, Service C accesses database β, and Services D and E access database χ. Notice that servers 1 and 2 provide service A. For a client running on box 2 to access database α, it need only call service A or B.

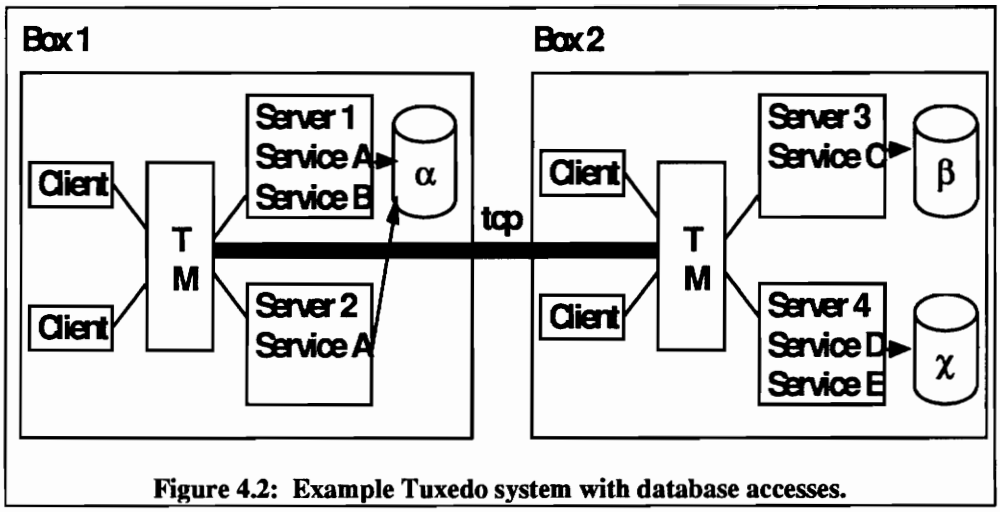

Figure 4.2: Example Tuxedo system with database accesses.

The reason Tuxedo refers to their paradigm as the "enhanced" client/server model is that service requests can be made synchronously or asynchronously. In synchronous mode, the client makes a service request and blocks until the reply is received. When the client returns from the function call, the reply is already in a buffer. When the client makes an asynchronous service request, the client does not block and must explicitly ask for the reply later.

While processing service requests, servers can make service requests for services provided by other servers. This commonly raises confusion regarding the terms "client" and "server" in Tuxedo. There are really two uses of the terms. A Tuxedo client process is a process which registers itself with Tuxedo as a client process. Similarly, Tuxedo recognizes server processes as server processes all the time. In the client/server model, the client is the requester in a service request and the server is the service provider of the service request. Therefore, when a server receives a service request and, in processing the service request, makes a service request, the server is the server in the first server request and a a client in the second service request; however, the process is always a server process.

4.3 Enterprise Transaction Processing (ETP)

As discussed in the overview of Tuxedo subsection of this section, Tuxedo has extended its OLTP support to Enterprise Transaction Processing (ETP) support. ETP allows global transactions to span databases on Personal Computers (PCs) and proprietary mainframe computers (Host Computers) in addition to UNIX boxes.[10] The ETP model involves three tiers of computers. The first tier comprises MS DOS, OS/2, Microsoft Windows, and UNIX workstations. The second tier is comprised of UNIX based Servers. The third tier comprises "mainframe class machines running proprietary operating systems and proprietary TP monitors." The second tier of UNIX servers plays the central role as that layer provides Transaction Monitoring "services to workstations and connections to mainframes."

The following figure from a book by Alex Berson on client/server architectures shows the layout of a Enterprise transaction processing network:

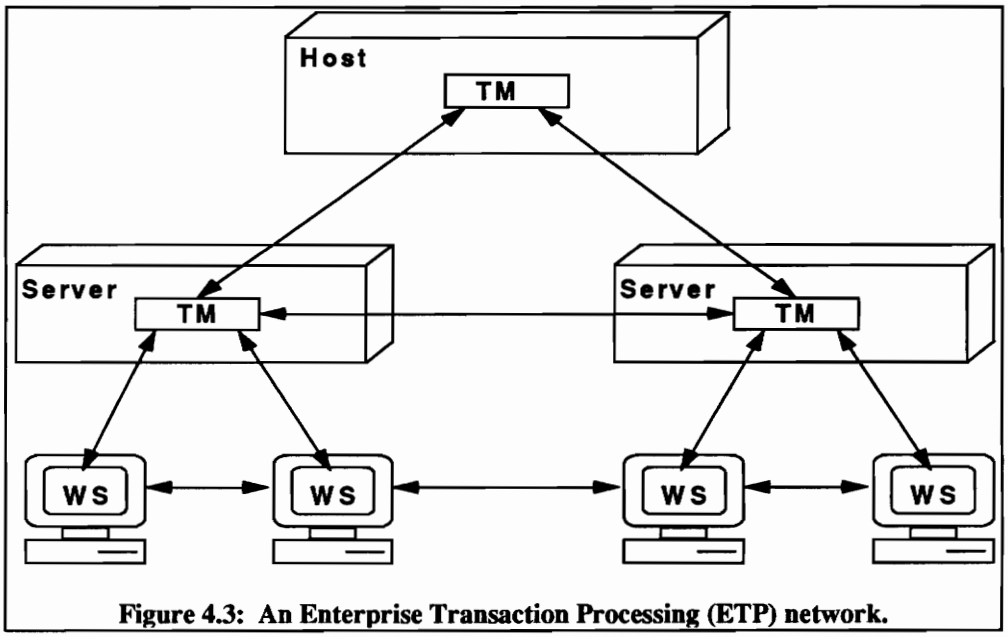

Figure 4.3: An Enterprise Transaction Processing (ETP) network.

In the network, Tuxedo provides for the transportation of messages through the network, management of the flow of transactions, the XA standard interface between the TM and RMs, management of the global transactions, and support for a data entry system for all supported workstations.

Russel Letson, in Systems Integration magazine discusses the recent movement of OLTPs onto personal computer (PC) local area networks (LANS).[11] Letson says that DOS historically was not as suited for OLTP processing as UNIX and the requirement for "concurrency and data-integrity mechanisms such as record locks, transaction logging, crash recovery and deadlock control" because OLTPs involve multiple users simultaneously accessing databases and DOS is limited to single users and tasks, has a 640K byte memory limit, and only has simple file locking mechanisms. However, better designs, such as client/server LAN architectures, and increased memory and processing are increasing the capability of spreading OLTP applications to PCs.

4.4 An Example of an Application Suited for Tuxedo

What kinds of applications are On-Line Transaction Processors suited for? An example of a large OLTP application was done for American Express Travel Related Services, Inc. (TRS)[12] In the early 1980's, American Express had over 30 million card holders around the world. While that was an indications of American Express's success, it also meant that TRS was receiving almost 3 million receipts per day. The flood of paper was resulting in increased costs, errors, and delays. TRS wanted to use electronic images to replace paper receipts, but there were no commercial products "big enough to perform the job."

In 1982, TRS began a project with Teknekkron Financial systems (which "eventually merged into TRW Financial Systems, Inc.") to develop a system which could handle the loads required by TRS. In 1988 the system became fully operational. TRS claims that the system is "the world's largest transaction processing system based on image processing and optical storage." The system "captures, digitizes, sorts, transmits, archives, and prints charge card images." Images are transferred between processes thorough machines

referred to as transports at the rate of 28 receipts per second per transport. Resolution for credit card images is 200 pixels per inch. TRS fills 5,000 optical disks per year with a volume of 100,000 receipts per disk. TRW's wide area image-transfer network can transmit 10 to 18 images per second and laser printers print 104,000 receipts per hour.

Cliff Dodd, vice president of billing and payment services for TRS' Consumer Card Group, who "spearheaded the systems integration team at TRS" describes how easily errors occurred in the old system. Receipts were photocopied for microfilm records. Account numbers were read by scanners with an 82% accuracy and charge amounts were entered by hand. Receipts were then placed in a collection of bins in an area half the size of a football field and read by 25 foot long readers/sorters. The receipts could go up to 8 times through the system until all the receipts for a user were together so that they could be returned to the user in the same envelope with their monthly statement. When a user questioned their bill, a person had to get out the microfilm and page through receipts until they found the one they were looking for. The task was very time consuming.

American Express, unlike VISA or Master Card, does not give a "float" or grace period (typically 25 days) before bills are due. Bills are due in full when the customer receives the bill. Therefore, the faster bills are sent the faster American Express gets its money and the less of its own money American Express needs to keep in funds to pay merchants. American Express could have stopped sending out copies of the credit card slips (referred to as country club billing), but decided to continue the practice in order to appeal to businessmen who use the receipts for expense accounts. American Express did not think it would be able to continue country club billing in a paper environment.

The following diagram is from the article on the American Express system:

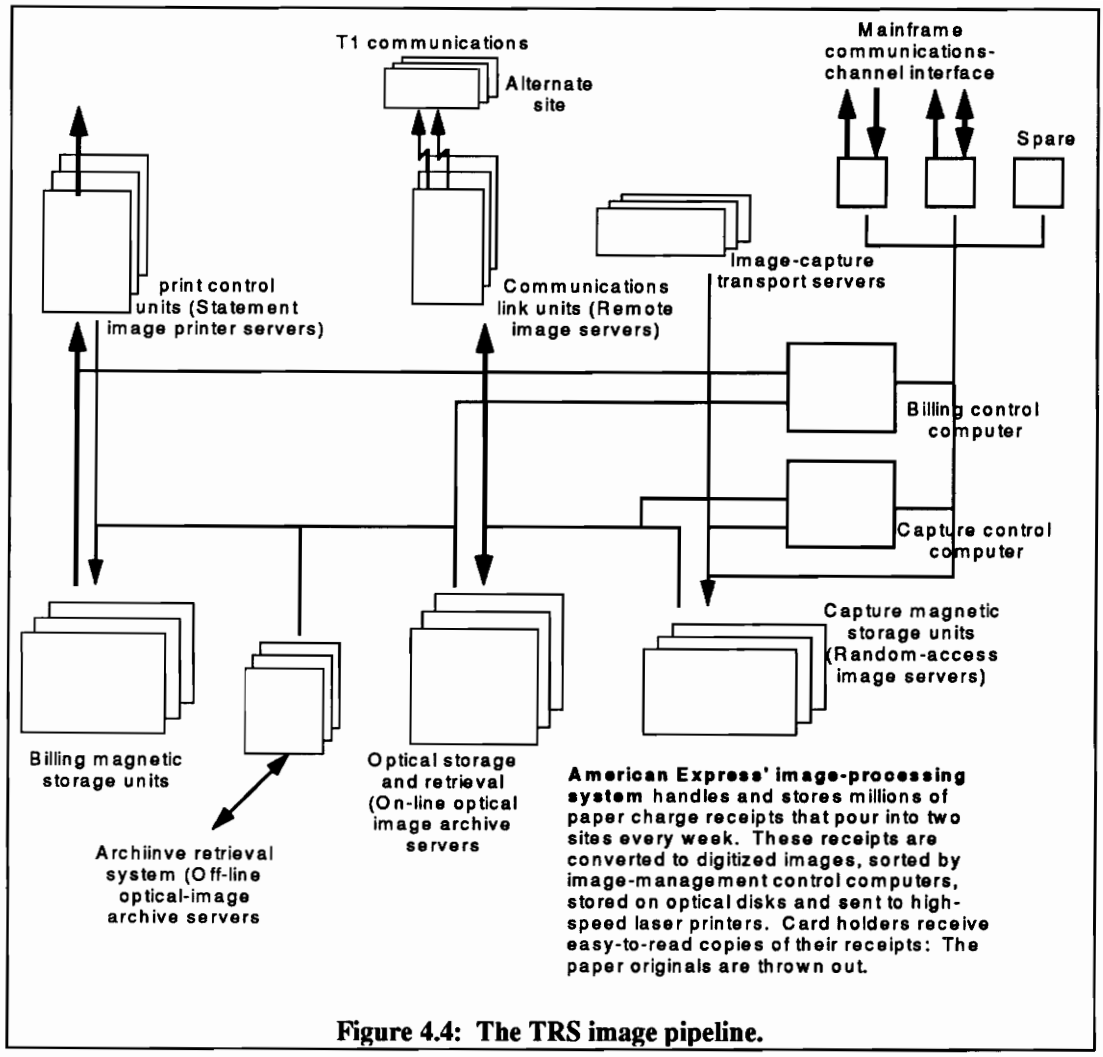

Figure 4.4: The TRS image pipeline.

TRS says they have reduced costs 25%. Forty percent of the savings are from the reduction of float periods and 60% are from the reduction of staff and equipment. For example, the number of people to handle receipts at each site has gone down from 210 to 3. Instead of nine paper sorting machines at each site, only two image capture servers are needed. When users question receipts, instead of searching through microfilm, operators can key in the account number and charge value and retrieve an image of the receipt.

4.5 Future Tuxedo Versions

According to Andy Huffman, vice president of USL's distributed-computing solutions group, USL is planning enhancements for the Tuxedo Transaction Monitor.[13] The plans are for Tuxedo to come in compliance with X/Open's XATMI and TX standards. XATMI "sets forth verbs to be used within transactions, such as requests for services. TX defines verbs to start and stop transactions. Compliance

with these standards, by both Tuxedo and competing monitors, means vendors can write tools knowing that they will work on any compliant monitor."

USL is planning a Dynamic Link Library (DLL) version of System /WS for its Microsoft Windows and OS/2 implementations. DLLs utilize memory more efficiently than the current staticly linked libraries used by Tuxedo. In staticly linked libraries, each process has its own copy of the library code. In DLLs, (referred to as shared libraries on UNIX), processes share a single copy of the code in memory. DLLs also have the benefit of allowing users of other products which use DLLs (like Microsoft's Visual Basic and Powersoft's PowerBuilder" to create applications which remotely access databases.

USL is also planning for Tuxedo to support SPX for use with Novell's NetWare product, the most common PC Local Area Network (LAN) protocol. Currently, PC users must have both TCP/IP and NetWare installed on their PCs, which requires "considerable amounts of memory." By supporting SPX, PCs will no longer be required to load TCP/IP as well.

In mid 1993, USL is planning to integrate Tuxedo with USL's implementations of the Open System Foundation's (OSF's) Distributed Management Environment (DCE) and Distributed Manager (DME) standards. Those standards are supported by other OLTPs, such as Transarc's Encina. DCE support will allow Tuxedo clients to use the Remote Procedure Call (RPC) syntax which "will offer Tuxedo's two-phase commit protection without the need to write separate code, as is usually necessary with RPCs." DME offers "graphical, object-oriented management of printers and backup devices -- and eventually Tuxedo applications -- on a TCP/IP network.

Other plans for Tuxedo include a COBOL API which will support the writing of both clients and servers under DOS, Windows, OS/2, and UNIX. Tuxedo is also planning a disk based queuing system for transactions, which is more reliable than the "current memory-based queuing of transactions." Finally, Tuxedo will provide a 3270 gateway, which will allow Tuxedo Unix applications to access mainframes through 3270 terminals "without having to change the mainframe applications."

Section V. Five Problems of Distributed Databases

C. J. Date describes five problems faced in distributed database processing in his book "An Introduction to Database Systems, Volume II."[14] The problems discussed by Date are query processing, update propagation, concurrency control, commit protocols, and catalog management. We will now discuss those five problems and how those problems are addressed in Tuxedo systems.

Tuxedo provides many distributed database management capabilities (e.g. two-phase commit). Tuxedo does not provide other basic capabilities of general distributed database management systems (e.g. a global catalog). As was previously discussed, vendors port Tuxedo to many platforms. Those vendors also provide tools which greatly expand Tuxedo's capabilities (including database management capabilities). For example Independence Technologies, Incorporated (ITI) provides an object oriented interface to XA compliant RDBMS (like Informix).

Database tools available for Tuxedo applications provide support for many of the database capabilities not provided by Tuxedo. This section does not discuss capabilities of vendor's database tools, but only the distributed database processing capabilities of Tuxedo. Keep in mind that many of the capabilities not provided by Tuxedo are provided by tools readily available for Tuxedo applications. Since Tuxedo is the most widely available OLTP for UNIX, it also has the most tools available for its applications.

We will now examine an example of the code required to process a global transaction in Tuxedo. The appendices discuss writing clients and servers in detail. In this example, we will use simple clients and servers and not go into great detail about how to write clients and servers. The emphasis of the example is the Tuxedo calls which control the global transaction, not how to write clients and servers.

The global transaction in our example will transfer money from an account record on one database to an account record on another. The strategy we will use in the transfer operation is to have one service, called "TRANSFER," which calls two services "WITHDRAW" and "DEPOSIT." In this case, the global transaction is coordinated by the TRANSFER service. In general, global transactions can be coordinated by clients or servers. We will refer to the client or server which is coordinating the transaction as the coordinating process since the transaction can be coordinated by a client or a server.

We will assume that the TRANSFER, WITHDRAW, and DEPOSIT services are on three different machines (with Tuxedo running on all three machines). We will also assume that the respective databases are on the same machine as the service which manipulates it and that all the databases are XA compliant. Here is the TRANSFER service. The amount to money to transfer will be passed into the services as a parameter.

```
TRANSFER( TPSVCINFO *msg )
{
    char *data;
    long len;
    long amount;

    data = msg->data;
    len = msg->len;
    amount = *((long *)data);

    /* begin global transaction */
    if ( tpbegin( 30, 0 ) == -1 )    {
        tpreturn( TPFAIL, 0L, NULL, 0, 0L );
    }

    /* withdraw money */
    if ( tpcall( "WITHDRAW", data, len, &data, &len, 0L ) == -1 ))    {
        < error >;
    }
    else if ( tpcall( "DEPOSIT", data, len, &data, &len, 0L ) == -1 ))    {
        < error >;
    }

    if ( !< error > )    {
        if ( tpcommit( 0 ) != -1 )    {
            tpreturn( TPSUCCESS, 0L, msg->data, msg->len, 0L );
        }
    }
    tpabort( 0 );
    tpreturn( TPFAIL, 0L, NULL, 0, 0L );
}
```

Figure 5.1: The TRANSFER Service.

The function tpbegin( ) informs Tuxedo that a global transaction is beginning and that all the database processing which follows is part of that transaction until the transaction is committed or rolled back. The coordinating process is the process which calls tpbegin( ). Therefore, TRANSFER is the coordinating process. The parameters for tpbegin( ) are the transaction time out value (in this case 30 seconds) and the flags (currently not used by Tuxedo). Transactions which time out are aborted. The tpcall( ) function invokes a service request. When tpcall( ) returns, the requested service has executed and the reply is in a buffer pointed at by "data." Please see the appendices for more details about service requests.

Since Transfer became the coordinating process by calling tpbegin( ), TRANSFER must also be the process which calls tpcommit( ) or tpabort( ) to indicate the success or failure (respectively) of the transaction. Once tpbegin( ) is called, all service requests are assumed by Tuxedo to be part of the transaction. If the services are accessing XA compliant databases, Tuxedo will automatically handle the two-phase commit protocol. Tuxedo begins the two-phase commit process when tpcommit( ) is called. When the tpcommit( ) call returns successfully, the changes to the database are permanent. Transactions are rolled back when the elapsed time is greater than the time out value in tpbegin( ) and when tpabort( ) are called.

We will now look at the WITHDRAW and DEPOSIT services which are called by TRANSFER:

```
WITHDRAW( TPSVCINFO *msg )
{
    long amount;

    amount = *((long *)data);

    < SQL code to withdraw money >

    tpreturn( TPSUCCESS, 0L, msg->data, msg->len, 0L );
}
```

Figure 5.2: The WITHDRAW Service.

```
DEPOSIT( TPSVCINFO *msg )
{
    long amount;

    amount = *((long *)data);

    < SQL code to deposit money >

    tpreturn( TPSUCCESS, 0L, msg->data, msg->len, 0L );
}
```

Figure 5.3: The DEPOSIT Service.

With these few lines of code, we could have moved money from an Informix database in Seattle to an Oracle database in France. The code is not concerned with the location of the databases. We will now discuss location and replication transparency in Tuxedo.

Three important concepts discussed by Date include agents, location transparency, and replication transparency. Date assumes that distributed database managers provide those capabilities for application programs. Tuxedo is not that high level and requires application participation in all three areas. As discussed earlier, there are also tools available to perform many of those functions. We will assume that we are only using Tuxedo.

Agents are processes which represent global transactions on different sites of a global transaction. Each site participating in the transaction must have at least one agent related to the transaction. Tuxedo requires application code for those agents. The agents in Tuxedo are servers which are located on the site of the database which is accessed by the server. At least one server must be provided by the application programmer on each site in the system where databases are located. The SQL code which manipulates the database for the client must be located in the server on the site of the database.

In section 6.1, TRANSFER was the code for the coordinating process who managed the transaction, and WITHDRAW and DEPOSIT were the agent code. WITHDRAW and DEPOSIT would need to be placed in servers which are configured to run on the same machine which contains the databases they access. TRANSFER could be located anywhere in the system since it does not directly access a database. In

Date's model, SQL code could be placed in TRANSFER which, using information in the system table, would manipulate remote databases. Tuxedo requires more explicit control on the part of the application.

In Date's model, the global catalog provides information to the distributed DBMS, including the location of the database tables in the system. Location transparency is provided since the SQL code is unaware of the location of the table. Furthermore, tables can be moved from a database on one site to another and the SQL code does not need to be changed since it looks in the global catalog for the location of the table. Tuxedo comes close to this capability, but uses another method to achieve location transparency.

In Tuxedo, instead of SQL code which accesses the global catalog to determine the site of a table, the coordinating process calls a service which has been configured in the Tuxedo configuration file to run on the same machine that the database is located. Since the coordinating process only calls the service by name, it too is unconcerned with the location of the database. In both cases, location transparency is provided since the code accessing the database is configured so to find the table. The difference is where that configuration takes place.

In replication transparency, multiple copies of records can be maintained without special application processing. When the value of a replicated record is desired, any copy of the record can be read. This reduces the number of messages which need to generated in the system. However, when replicated records need to be updated, all copies of the record needs to be updated. In Date's model, the distributed database manager provides direct support for replication transparency. Tuxedo does not directly provide this capability. Tuxedo does provide capability for the application to reliably maintain replicated records. Maintaining replicated records in Tuxedo will be discussed further in the next two subsections of this section.

5.1 Query Processing

When a client needs to perform a query on a remote database, a service must be written and placed on the remote site which performs the query and returns the results of the query as a message to the client. The service which is to perform the query can open a cursor, read the records, build a message, and return the results of the query to client. In XA, the server which opens the cursor is the only process allowed to use and close the cursor. To make reading the queries easier to read in the example code in this section, the cursor code will be omitted and queries will be written as simple select statements. Here is an example of a service which will return perform a query with one result:

[allocate message as size of record + message overhead]

SELECT Address, City, State, Zip
FROM EmployeeInfo
WHERE Name = 'Smith, John'

[place result in message]
[return message]

For the case of a single result with a known size (or any known or maximum number of results), this method works well. One problem faced by services is if a query could result in an unknown number of matches. For example, what if the application allocates a message large enough for 10 records and the query returns 11. The application has to make a decision as to how to allocate memory for the message. One possible method the service could use is to reallocate the block for each record returned by the query.

A variation of this method would be to allocate n records at a time. The application could also lock the table and count the records before allocating the buffer and reading the records.

[lock table]

SELECT count(*)
FROM EmployeeInfo
Where Name = 'Smith, John'

[allocate message as (record size * count) + message overhead]

SELECT Address, City, State, Zip
FROM EmployeeInfo
WHERE Name = 'Smith, John'

[unlock table]
[return message]

This method is not very graceful since the records are accessed twice to read them once. Unfortunately, there is no simple solution to this problem and the application needs to determine the best method relying on its knowledge of the application.

A bigger problem in Tuxedo is joins. If joins are to be performed on tables in the same database, the service which does the local query can let the local database manager perform the join. The problem is if a join is to be performed on tables in different databases. In this case, a service must be written which performs the query locally for each database and returns the results as a message to the service coordinating the transaction. The service must then manually perform the join. Clearly, Tuxedo is not designed to perform complicated joins across distributed databases. Again, tools available for Tuxedo applications address this capability.

5.2 Update Propagation

Update propagation involves updating replicated records. As discussed before, Tuxedo does not support replication transparency and the application needs to manage update propagation for replicated records. When there are no network problems or down sites, the update of replicated records is relatively straight forward. The service coordinating the transaction calls a service on each site containing a copy of the replicated record.

The coordinating service passes the new value of the record as data in the service request. The service on each site attains an exclusive lock on the record and changes the value of the replicated record to its new value. If all the service requests are successful, the global transaction is committed and the new value becomes permanent at each site. If the service request on any site is unsuccessful, either in locking or updating its copy, that service fails and the transaction is rolled back on all the sites so that the new value does not become permanent on any site in the network.

If a site is unavailable (e.g. a network failure results in loss of communications to a site), then update becomes more tricky. In Tuxedo, the application needs to decide how to handle this condition. One solution is to respond to the client that the update failed. Date suggests two methods that distributed

database managers can use when sites are unavailable. These methods can also be used by Tuxedo applications. Although in Tuxedo, the application needs to handle the code.

One method suggested by Date is to keep a log of the sites which were not updated during the transaction and update them when they become available. This can be accomplished in Tuxedo by writing the updates to a log. Tuxedo provides the capability to execute a script when a server which is down comes back on line. A server on a site which went down would be considered down and could therefore use this capability. The script is executed before the server receives new service requests. In this case, the script executed when the server comes on line could read the log and process the missed updates before processing any service requests. Since the log would be read sequentially, the copies of replicated records on the site would have the correct value before the server processes any service requests.

The other method suggested by Date is to have a "primary" copy of replicated records. This method could be implemented more easily in Tuxedo since, using the script method used in the last example, the script could simply query the value of the primary copy before processing service requests. While this method is easier to implement, it is also less reliable since problems arise if the site of the primary copy is the site which becomes unavailable.

5.3 Concurrency Control

We will now discuss serializability, two phase locking, and global deadlocks. Serializability means the interleaved executions of multiple transactions will not violate the integrity of data in the database. For example, suppose transaction 1 reads a value from a field in a record in a database, adds one to it, and writes the record back to the database. Transaction 2 reads the value, subtracts one from it, and writes the record back to the database. If the transactions are serializable, and each transaction is executed on the same record, the value would be the same after both transactions executed as it was before the transactions executed since it had one both added to it and subtracted from it.

We will now examine how the interleaved execution of the two transactions might return different results, and therefore not be serializable. Suppose that the initial value of the record is seven and the following execution sequence occurs:

(T1) : read value (7)
(T2) : read value (7)
(T1) : add 1 to value (8)
(T2) : subtract 1 from value (6)
(T1) : write value back to field (8)
(T2) : write value back to field (6)

The final value of the field is 6. However, suppose this sequence were executed:

(T2) : read value (7)
(T1) : read value (7)
(T2) : subtract 1 from value (6)
(T1) : add 1 to value (8)
(T2) : subtract 1 from value (6)
(T1) : write value back to field (8)

This time, the final value of the field is 8. If both sequences are legal, these transactions would not be serializable since different interleaved sequences violate the integrity of the data. This problem is referred to as the lost update problem.

Date states that if two-phase locking is used by transactions, then they will be serializable. In two-phase locking, a transaction never accesses records without locking them. Also, once a record is unlocked, then no more records are allowed to be locked by the transaction. Not locking records can clearly lead to a lack of serializability. The reason for the constraint that no records should be locked after one is unlocked is that if a record is unlocked and then locked again, other transactions can change the value in between the unlock and lock, which can lead to a lack of serializability. If these rules are not followed, that does not imply that the transaction is not serializable. However, if the rules are followed, it does imply that the transaction is serializable.

Tuxedo does not support the concept of "global locks." Locks can be performed by the services which are accessing the local database manager. The easiest way that two-phase locking can be implemented is by having the services which call the local database manager lock the appropriate record(s), table(s), and database(s) and not release the lock until the transaction is committed or rolled back. In fact, many RDBMS (like Informix) do not allow objects which are locked during a global transaction to be unlocked until the transaction terminates (via commit or rollback). Furthermore, the locks are automatically released when the transaction terminates.

The next topic of this subsection is global deadlock. Since Tuxedo only manages calling application code resident on the site of each database, it does not know what operations the service is performing on the local DBMS. Tuxedo is therefore not capable of detecting global deadlock. The vehicle that Tuxedo uses to control global deadlock is timers. As shown in the example, the application sets a timer when it calls tpbegin( ). After the specified time has elapsed, the transaction is automatically rolled back. If a deadlock occurs, the first transaction which times out will be rolled back and its locks will be released which allows another transaction to acquire its locked objects. Tuxedo will clearly not perform well on a system where global deadlocks commonly occur.

We will now look at two sequences of transactions in Tuxedo in order to examine how blocking and deadlock are resolved in the sample transactions in order to provide concurrency control. We will have two records (A and B), two services to read A and B (SR1 and SR2 respectively), two services to update A and B (SU1 and SU2 respectively, two transactions (T1 and T2) which operate on the two records, and two coordinating processes to manage T1 and T2 (CP1 and CP2 respectively). Here are the transactions:

T1:
[CP1 calls tpbegin( )]
[CP1 calls SR1]
[SR1 calls DBMS to acquire shared lock on A]
[SR1 reads the value of A using SQL]
[SR1 returns value of A to CP1]
[CP1 calls SR2]
[SR2 DBMS to acquire shared lock on B]
[SR2 reads the value of B using SQL]
[SR2 returns value of B to CP1]
[CP1 calls tpcommit( )]
[Tuxedo releases locks (2PC not needed)]

T2:
[CP2 calls tpbegin( )]
[CP2 calls SU2 with new value for B]
[SU2 calls DBMS to acquire exclusive lock on B]
[SU2 updates the value of B using SQL]
[SU2 returns successful update of B to CP2]
[CP2 calls SU1]
[SU1 calls DBMS to acquire exclusive lock on A]
[SU1 updates the value of A using SQL]
[SU1 returns successful update of A to CP2]
[CP2 calls tpcommit( )]
[Tuxedo performs 2PC and releases locks]

We will first look at a sequence of these transactions where T2 will block because T1 has locked A and B. When T1 completes, T2 will continue.

T1	time	T2
[CP1 calls tpbegin( )]	t1	
[CP1 calls SR1]	t2	
[SR1 calls DBMS to acquire shared lock on A]	t3	
[SR1 reads the value of A using SQL]	t4	
[SR1 returns value of A to CP1]	t5	
[CP1 calls SR2]	t6	
[SR2 calls DBMS to acquire shared lock on B]	t7	
	t8	[CP2 calls tpbegin( )]
	t9	[CP2 calls SU2 with new value for B]
	t10	[SU2 calls DBMS to acquire exclusive lock on B] SU2 blocks since B has shared lock
[SR2 reads the value of B using SQL]	t11	
[SR2 returns value of B to CP1]	t12	
[CP1 calls tpcommit( )]	t13	
[Tuxedo releases locks (2PC not needed)]	t14	
	t15	[SU2 acquires exclusive lock on B (just released)]
	t16	[SU2 updates the value of B using SQL]
	t17	[SU2 returns successful update of B to CP2]
	t18	[CP2 calls SU1]
	t19	[SU1 calls DBMS to acquire exclusive lock on A]
	t20	[SU1 updates the value of A using SQL]
	t21	[SU1 returns successful update of A to CP2]
	t22	[CP2 calls tpcommit( )]
	t23	[Tuxedo performs 2PC and releases locks]

We will now look at a sequence of these transactions where T1 and T2 will deadlock. Tuxedo will rollback T2, and T1 will complete.

T1	time	T2
[CP1 calls tpbegin( )]	t1	
[CP1 calls SR1]	t2	
[SR1 calls DBMS to acquire shared lock on A]	t3	
[SR1 reads the value of A using SQL]	t4	
[SR1 returns value of A to CP1]	t5	
	t6	[CP2 calls tpbegin( )]
	t7	[CP2 calls SU2 with new value for B]
	t8	[SU2 calls DBMS to acquire exclusive lock on B]
	t9	[SU2 DBMS to acquire exclusive lock on B]
	t10	[SU2 updates the value of B using SQL]
	t11	[SU2 returns successful update of B to CP2]
[CP1 calls SR2]	t12	
[SR2 calls DBMS to acquire shared lock on B]	t13	
SR2 blocks because T2 has exclusive lock on B		
	t14	[CP2 calls SU1]
	t15	[SU1 calls DBMS to acquire exclusive lock on A] SU1 blocks because T1 has shared lock on A
!!! deadlock !!!		
	tn	T2 times out Tuxedo rolls back T2 & releases locks
[SR2 acquires shared lock on B]	tn+1	
[SR2 reads the value of B using SQL]	tn+2	
[SR2 returns value of B to CP1]	tn+3	
[CP1 calls tpcommit( )]	tn+4	
[Tuxedo releases locks (2PC not needed)]	tn+5	

5.4 Commit Protocols

Tuxedo's commit protocol is the two-phase commit protocol. The example code in subsection 6.1 showed how the application interfaces to Tuxedo in order to use the protocol. There are three topics which will be discussed here related to the two-phase commit protocol. The first is when tpcommit( ) returns. The second topic relates to error handling during two-phase commit operations. The third topic discusses optimizations used to reduce the number of messages required to perform the two-phase commit process.

We previously discussed that the process coordinating the global transaction calls tpbegin( ) to begin a global transaction. The process then calls tpcommit( ) when it wants Tuxedo to begin the two-phase commit process. The application has two choices as to when tpcommit( ) should return. For maximum reliability, the application should have tpcommit( ) return when the entire two-phase commit processing has completed. If this method is used, the application knows with maximum reliability that the transaction has successfully updated each database.

The alternative option is that tpcommit( ) can return after the end of the first phase messages all complete successfully. Tuxedo completes the two-phase commit protocol behind the scenes and the application is not notified if a problem occurs in the second phase. If the first phase messages all complete successfully, it is rare that a second phase message would fail (e.g. if a communications link is lost). Returning after the first phase greatly increases the speed of the application. If losing a few updates over a long period of

time is acceptable, than this method could be used to increase performance. If it is unacceptable to ever lose an update, the first method should be used.

The next subject is how rollbacks can be triggered by errors. If the time specified in the tpbegin( ) call elapses and the transaction is not yet complete, then the transaction is rolled back. If the process coordinating the transaction calls a service to perform a database operation and an error occurs in the database service, the service returns that it failed an error is returned to the coordinating process who initiates a rollback. Similarly, if a Tuxedo error occurs during a service request from the coordinating process, the service request fails and the coordinating process initiates a rollback. If an error while tpcommit( ) is executing the two-phase commit protocol, the transaction is rolled back and tpcommit( ) returns an error.

Finally, the coordinating process can request a rollback at any time during the transaction before tpcommit( ) is called. For example, consider the TRANSFER, WITHDRAW, and DEPOSIT operations. Suppose if in the WITHDRAW service, the account balance went to -$100. The database operation would not fail because the user has an account with a balance. However, we may not want to allow accounts with negative balances. The way the transaction would be rolled back in Tuxedo is that the WITHDRAW service would return an application error code to the TRANSFER service who would call tprollback( ) to roll back the transaction. No process, other than the coordinating process, can instruct Tuxedo to roll back the transaction. Services can return application errors which cause the coordinating process to initiate a rollback. If another process attempts to roll back the transaction, Tuxedo will return an error to the requesting process.

The next subject of this subsection is about automatic message reduction techniques used by Tuxedo during execution of the two-phase commit protocol. We will not discuss a comprehensive list of what Tuxedo does to reduce messages, but we will discuss a couple of techniques Tuxedo uses. The first method to reduce messages is that Tuxedo may reduce a two-phase commit to one phase commit protocol. For example, if the process only accesses a single database manager during a global transaction, Tuxedo will automatically instruct the RDBMS to perform a simple one phase commit operation when the coordinating process calls tpcommit( ).

Another strategy used to reduce messages is that Tuxedo can only send the two-phase commit messages to sites on which databases were actually updated. Since Tuxedo does not know what operations are performed by a server when it accesses a database, the database manager must inform Tuxedo that no updates were made during the transaction branch. The RDBMS can use the XA protocol for this purpose. If the RDBMS does inform Tuxedo that no updates were made, Tuxedo will not send any of the first or second phase messages to the RDBMS.

The final topic of discussion in this section is recovery. The two failure conditions which can occur in which the transaction would need to be aborted are that the coordinating process can fail or services participating in the transaction can fail. A service participating in the transaction failing includes: the server the service is in crashes, the machine the service is on can crashes, or a network problem causes communications with the server to be lost . If any of these conditions occur, Tuxedo will return an error to the coordinating process to inform it that the service request failed. For example, in the TRANSFER service, tpcall( ) would return a -1 and the application can look in a global variable for the specific error condition. The coordinating process then calls tprollback( ) to trigger a rollback on all the databases which are participating in the transaction.

If the coordinating process crashes, Tuxedo triggers the rollback automatically. Remember, a Transaction Manager (TM) is running on each site in the Tuxedo system. When the coordinating process calls tpbegin( ), it is the Transaction Manager who is performing the functionality discussed (e.g. coordinating

the two-phase commit protocol). If the coordinating process crashes during a global transaction, the TM knows that the process crashed and that the coordinating process was coordinating a transaction. The TM would, when it detects the coordinating process crashed, initiate a rollback.

5.5 Catalog Management

Tuxedo does not have a global catalog. The discussions in the other sections showed how queries can be performed on local databases by services, but the responses are returned as messages. The application could use a database for tables dedicated to the management of the application. For example, a database could contain a table containing a list of sites where replicated copies of data are stored. This database would simply be another local database manager to Tuxedo and only the application would consider it a global catalog.

Section VI. Tuxedo and Other OLTPs

We will now look briefly at the similarities of and differences between of Tuxedo and the other major OLTPs targeting the UNIX environment. This will not be a detailed, technical look, but a look at the basic similarities and differences. Included in the discussion are some of the capabilities each product advertises for their platform.

The first product is Transarc's Encina, which uses the most advanced technology of the OLTPs. Transarc's Encina is based on OSF DCE. The use of advanced technology is also Transarc's weakness as it is slow getting to the market. Top End is more advanced than Tuxedo, but it is also new to the market. IBM is planning to implement CICS (its mainframe OLTP) on UNIX. The primary beneficiaries of CICS on UNIX will be shops which have CICS applications and CICS trained programmers.

6.1 Transarc Encina Transactions Processing

A report by Datapro, in Datapro Reports on UNIX Systems & Software, discusses the Encina Transaction Processing Software system.[15] Encina is based on the OSF's DCE specification. DCE is a standard comprised of cutting edge technology, and it is not yet clear whether DCE itself will be accepted as an industry standard. The success of Encina will depend greatly on the success of DCE. We will not discuss DCE extensively here, as that could be another complete paper in itself. Encina uses "DCE Remote Procedure Call (RPC), Authentication Service, DCE Cell Directory Services, and DCE Threads." Because of the cutting edge nature of DCE and Encina, Encina is one to three years away from delivering "production model products."

We will now look at the architecture of Encina. The next diagram, from the Datapro Article shows the layers which comprise Encina. Tier 1 is the base platform and Tier 2 are products Transarc sells based on the platform.

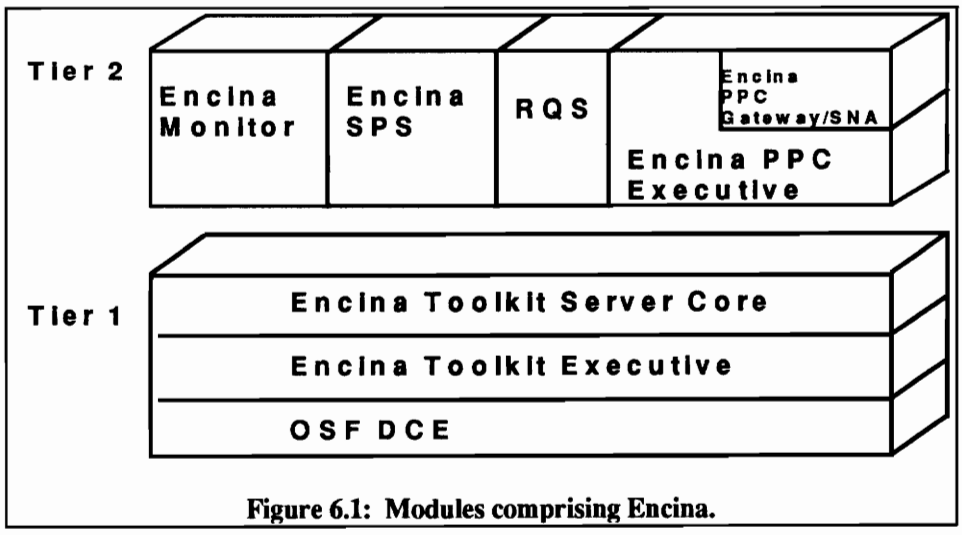

Figure 6.1: Modules comprising Encina.

Encina is built on top of OSF DCE. On top of DCE is the Encina Toolkit Executive, which extends DCE services to "enable transactional client/server processing." The next layer, the Toolkit Server Core, supports data locking," a recoverable storage system based on write-ahead logging," and the XA Interface for TM coordination of RMs during global transactions. The top layer is products which are provided by Encina, including the TM itself, a Structured File Server (SFS), a Recoverable Queuing Service (RQS), and Peer-to-Peer Communications (PPC).

We will first discuss the similarities between Encina and Tuxedo, and then the differences. Both are on-line transaction processors whose applications are developed under the client/server model. Like Tuxedo, Encina is based on X/Opens DTP model, supports the XA interface between its TM and RMs running with Encina, and uses the two-phase commit protocol. While Encina "does not use object-oriented code, its design is based on object-oriented philosophy."

One of the major differences between Encina and Tuxedo (between Encina and virtually all current OLTPs) is that Encina attempts to support horizontal in addition to vertical compatibility. In vertical compatibility, different layers (e.g. a TM and RM) have a standard interface. In horizontal compatibility, different pieces of the same level have common interfaces and are thus interchangeable. While X/Open eventually hopes to support horizontal compatibility, DCE is attempting to do that today.

Another fundamental difference between Encina and Tuxedo is that by using DCE Threads (as mentioned earlier), Encina supports multi-threading. Tuxedo is strictly single threaded. The advantage of multi-threading is that multiple threads of control can share the same context (e.g. address space, open files, and resource locks). Since the threads share the same context, they can work on the same problem and exchange information in memory, thereby virtually eliminating communications overhead. The disadvantage of multi-threading is that it increases the complexity of the programs. When multiple processes are sharing resources, they must manage contention while accessing those resources. In multi-threaded programs, potential contention greatly increases since the threads are referencing the same libraries and memory. Non-reentrant libraries and common buffers must be designed so that the threads do not interfere with each other.

Tuxedo has been in the market for years and is currently installed in over 1,000 sites, which gives Tuxedo an advantage over Encina. On the other hand, Tuxedo was developed before there were many industry standards for the OLTP market. That means that Tuxedo is more dependent on UNIX and lower level constructs than newer OLTPs like Encina, which are layered so that only the lower levels of the environments are dependent on operating system constructs. This is a clear advantage for Encina over Tuxedo.

6.2 Top End

Top End is described in the Top End Product Overview by NCR.[16] "The primary objective of Top End is to bring mainframe class DTP capabilities to open systems." Like Tuxedo, Top End is based on X/Open's DTP model, supports the XA interface between its transaction manager and RMs in the system, and uses the two-phase commit protocol. Top End applications are also written using the client/server model. NCR says that they have over 100 years of transaction processing history (beginning with a cash register) and over 25 years of DTP history (beginning with banking applications).

One difference between Top End and Tuxedo is that Top End supports remote clients. Tuxedo client processes can be running anywhere in a distributed Tuxedo system, but Tuxedo's Transaction Manager must be running on each site in the network. While Top End supports Tuxedo's model, Top End allows

clients to be running on remote sites (a machine where a TM is not running), and the client process can still participate in a global transaction. Top End also provides international support, including both informational and error logging conversions to French, Spanish, and English.

Top End provides a Communication Manager Extension to the DTP model. Some of the benefits of this model is that Communication Managers are shared resources not fully represented by the DTP model because the XA specification is not sufficient to represent a communications resource, which is capable of being "both superiors and subordinates in distributed transactions." The communication manager also allows heterogeneous TMs to coordinate global transactions together. Since communication managers can be either superior or subordinates, global transactions can be initiated by either TM.

The next figure, from the Top End Overview document, shows Top End's Communication Manager Extension to the DTP model.

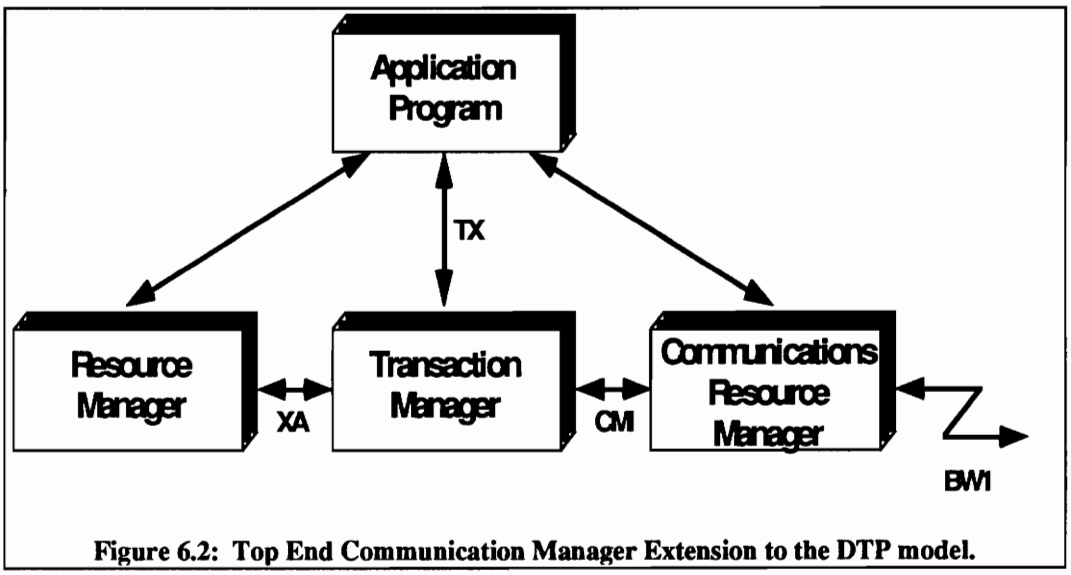

Figure 6.2: Top End Communication Manager Extension to the DTP model.

6.3 CICS

According to Advanced Network Computing magazine, IBM is offering both hardware and software to get into the UNIX OLTP market.[17] The hardware is RS/6000 series models. The software is its Customer Information and Control System (CICS) transaction processing system. CICS will be ported to the RS/6000 series machines, OS/2, AS/400, and MVS based on DCE and Encina. Like Tuxedo, CICS is compatible with the XA Specification and both support client/server architected applications.

The Gartner Group Industry Service compares CICS to Tuxedo in the InSide Gartner Group This Week magazine.[18] CICS is an advantage for companies with staff trained to program CICS and who want to integrate UNIX CICS applications with existing CICS mainframe applications. While Tuxedo supplies a CICS gateway, "CICS has more powerful and easier access to TP applications, e.g., the thousands of CICS/MVS applications." CICS for UNIX is "95 percent compatible with CICS/MVS applications and familiar to IS staffs that are trained in CICS." This also means that UNIX CICS "applications can easily inter-operate with other versions of CICS."

A disadvantage to CICS, compared to Tuxedo, is that CICS does not currently support PC DOS and Microsoft Windows clients in their applications. CICS provides an advantage for current IBM shops which have CICS programmers and applications, but Tuxedo will be more appealing to UNIX developers since it is more natural to that environment. If there is no reason to use CICS, then Tuxedo (or another OLTP) may be a better alternative since using Tuxedo is more natural to UNIX programmers.

Conclusion

This report identifies how Tuxedo fits into the scheme of distributed database processing. Tuxedo is an On-Line Transaction Processing (OLTP) system. Tuxedo was studied because it is the oldest and most widely used transaction processing system on UNIX. That means that it is established, extensively tested, and has the most tools available to extend its capabilities. The disadvantage of Tuxedo is that newer UNIX OLTP systems are often based on more advanced technology. For this reason, other OLTPs were examined to compare their additional capabilities with those offered by Tuxedo.

As discussed in Sections I and II, Tuxedo is modeled according to the X/Open's Distributed Transaction Processing (DTP) model. The DTP model includes three pieces: Application Programs (APs), Transaction Monitors (TMs), and Resource Managers (RMs). Tuxedo provides a TM in the model and uses the XA specification to communicate with RMs (e.g. Informix). Tuxedo's TX specification, which defines communications between the APs and TMs is also being considered by X/Open as the standard interface between APs and TMs. There is currently no standard interface between those two pieces. Tuxedo conforms to all X/Open's current standards related to the model.

Like the other major OLTPs for UNIX, Tuxedo is based on the client/server model. Tuxedo expands that support to include both synchronous and asynchronous service calls. Tuxedo calls that extension the enhanced client/server model. Tuxedo also expands their OLTP support to allow distributed transactions to include databases on IBM compatible Personal Computers (PCs) and proprietary mainframe (Host) systems. Tuxedo calls this extension Enterprise Transaction Processing (ETP). The name enterprise comes from the fact that since Tuxedo supports database transactions supporting UNIX, PCs, and Host computers, transactions can span the computer systems of entire businesses, or enterprises.

Tuxedo is not as robust as the distributed database system model presented by Date. Tuxedo requires programmer participation in providing the capabilities that Date says the distributed database manager should provide. The coordinating process is the process which is coordinating a global transaction. According to Date's model, agents exist on remote sites participating in the transaction in order to handle the calls to the local resource manager. In Tuxedo, the programmer must provide that agent code in the form of services.

Tuxedo does provide location transparency, but not in the form Date describes. Date describes location transparency as controlled by a global catalog. In Tuxedo, location transparency is provided by the location of servers as specified in the Tuxedo configuration file. Tuxedo also does not provide replication transparency as specified by Date. In Tuxedo, the programmer must write services which maintain replicated records.

Date also describes five problems faced by distributed database managers. The first problem is query processing. Tuxedo provides capabilities to fetch records from databases, but does not provide the capabilities to do joins across distributed databases. The second problem is update propagation. Tuxedo does not provide for replication transparency. Tuxedo does provide enough capabilities for programmers to reliably maintain replicated records. The third problem is concurrency control, which is supported by Tuxedo. The fourth problem is the commit protocol. Tuxedo's commit protocol is the two-phase commit protocol. The fifth problem is the global catalog. Tuxedo does not have a global catalog.

The other comparison presented in the paper was between Tuxedo and the other major UNIX OLTPs: Transarc's Encina, Top End, and CICS. Tuxedo is the oldest and has the largest market share. This gives

Tuxedo the advantage of being the most thoroughly tested and the most stable. Tuxedo also has the most tools available to extend its capabilities. The disadvantage Tuxedo has is that, since it is the oldest, it is based on the oldest technology.

Transarc's Encina is the most advanced UNIX OLTP. Encina is based on DCE and supports multi-threading. However, Encina has been slow to market and has had stability problems because of its advanced features. Also, since Encina is based on DCE, its success is tied to the success of DCE. Top End is less advanced than Encina, but more advanced than Tuxedo. It is also much more stable than Encina. However, Top End is only now being ported from the NCR machines on which it was originally built. CICS is not yet commercially available. CICS is good for companies with CICS code to port to UNIX and CICS programmers who are already experts. The disadvantage to CICS is that companies which work with UNIX already and do not use CICS will find the interface less natural than Tuxedo, which originated under UNIX.

Appendix I. Building and Running a Tuxedo Application

We will now look at the code a Tuxedo application programmer would need to write to create a client and server for a simple Tuxedo application. We will then look at how to build the executables, set the environment variables, and run the Tuxedo application. Tuxedo functions called by the client and server will be discussed in detail in later sections of the document. The purpose of the example is to demonstrate how simple it is to create and run a Tuxedo application.

A1.1 A Sample Tuxedo Application

Our Tuxedo application will have a Tuxedo client, called "average," who will be run from the command line and passed three numbers as arguments. The client will call a server, (named "server") who provides a service called "AVERAGE." The "AVERAGE" service calculates the average of the three numbers and returns the result to the client, who will print the result.

A1.1.1 A Tuxedo Client

Here is the code for the client:

```
#include <stdio.h>
#include <atmi.h>

int main( int argc, char *argv[ ] )
{
    long *buf, len = 3 * sizeof( long );

    /* connect to tuxedo */
    if ( tpinit( NULL ) != -1 )    {
        /* allocate a buffer for the service request */
        if (( buf = (long *)tpalloc( "CARRAY", NULL, len )) == 0 )    {
            printf( "Could not allocate a buffer for the service request \n" );
        }

        else    {       /* fill in the buffer and make the service request */
            buf[ 0 ] = atol( argv[ 1 ] );
            buf[ 1 ] = atol( argv[ 2 ] );
            buf[ 2 ] = atol( argv[ 3 ] );

            /* request service "AVERAGE", print the result of the call, and free the buffer */
            if ( tpcall( "AVERAGE", (char *)buf, len, (char **)&buf, &len, 0 ) == -1 )    {
                printf( "Service request failed \n" );
            }
            else    {       /* note: data is  returned in the same buffer pointer */
                printf( "Average ( %s, %s, %s ) = %ld \n", argv[ 1 ], argv[ 2 ], argv[ 3 ], buf[ 0 ] );
            }
            tpfree( (char *)buf );
        }
        /* terminate the Tuxedo connection and exit */
        tpterm( );
    }
    exit( 1 );
}
```

Figure A1.1: A sample Tuxedo client.

A1.1.2 A Tuxedo Server

Now, let us look at the source code for the server which will process the "AVERAGE" service request. Unless explicitly changed, the name of the function which processes a service request is the same name as the name of the service. In this document, we will assume that name of the service and the function which processes the service are the same. Notice that there is no "main" program in the server. When servers are built, Tuxedo provides the main program and the applications programmer provides the functions which process the service requests. In the sample server code, the userlog( ) function writes messages into a central log. The client could also have called userlog( ).

```
#include <stdio.h>
#include <atmi.h>
#include <userlog>

AVERAGE( TPSVCINFO *msg )
{
    long result, *buf;

    /* assign a pointer to the data */
    buf = spMessage->data;

    /* calculate the average of the numbers and log the result */
    result = buf[ 0 ] + buf[ 1 ] + buf[ 2 ];
    userlog( "Service AVERAGE: the average of %ld, %ld, and %ld, is %ld",
                buf[ 0 ], buf[ 1 ], buf[ 2 ], result );

    /* return the result */
    buf[ 0 ] = result;
    tpreturn( TPSUCCESS, 0L, msg->data, sizeof( long ), 0L );
}
```

Figure A1.2: A sample Tuxedo server.

A1.2 Running Tuxedo Applications

We will next look at how to compile the client and server source code and create the executables. The steps involved are initializing environment variables that Tuxedo will use and creating a make file which will build the executables. We will then discuss how to run Tuxedo systems. First, we need to create and compile a configuration file that Tuxedo will use at run-time which has information such as the names and number of copies of servers to run and other information that Tuxedo will need to know to run the system. We will then look at how to boot the Tuxedo system and run the client.

A1.2.1 The Tuxedo Environment Variables

There are environment variables which must be set to compile and run Tuxedo programs. The standard way to set the variables is to create a shell script (e.g. .tuxprofile) which initializes the environment. The following is an example of a .tuxprofile written for the Korn Shell:

```
ROOTDIR=/tux
APPDIR=`pwd`
PATH="${PATH}:${ROOTDIR}/bin"
TUXCONFIG=`pwd`/tuxconfig
UBBCONFIG=`pwd`/UBBconfig
export ROOTDIR APPDIR TUXCONFIG UBBCONFIG
```

Figure A1.3: A sample Korn shell script to initialize the environment.

This script says that the Tuxedo libraries, include files, binaries, etc. can be found in sub-directories of the /tux directory. The application software is in the present working directory (pwd) at the time the script is executed. The directory /tux/bin is added to the path. The tuxedo configuration files are named tuxconfig and UBBconfig (which will be explained later) and are located in the pwd. The environment variables are all then exported so that Tuxedo can find them.

A1.2.2 A Makefile to Create Clients and Servers

The following makefile could be used to create the server and client for our system:

```
ROOTDIR=/tux
INCDIR=$(ROOTDIR)/include
BINDIR=$(ROOTDIR)/bin
CFLAGS=-I$(INCDIR) -g

CSRC = server.c average.c
COBJ = $(CSRC:.c=.o)
OBJS = server average

all: $(OBJS)

server: server.o
    $(BINDIR)/buildserver -v \
        -o server \
        -s AVERAGE \
        -f server.o \
        -f /lib/libm.a

average: average.o
    $(BINDIR)/buildclient -v \
        -o average \
        -f server.o
```

Figure A1.4: A sample makefile for a Tuxedo system.

The commands buildserver and buildclient are used to create servers and clients respectively. Notice that when "server" is built, the name of the service it provides (AVERAGE) is specified as a "-s" option. Also notice that our "AVERAGE" function which is in the file server.c is built into the server, and no main is specified.

A1.2.3 The Tuxedo Configuration File

Tuxedo needs a configuration file to give it the information it needs to boot and run the system. The configuration file will be discussed more extensively in Section II. The following configuration file (normally called UBBconfig) could be used to run our application.

```
*RESOURCES
IPCKEY          72145
UID             827
GID             27
MAXACCESSERS    25
MAXSERVERS      10
MASTER          SITE1
PERM            0660
MODEL           SHM
LDBAL           Y
*MACHINES
bart LMID=SITE1
    ROOTDIR="/tux"
    APPDIR="/bart/user1"
    TUXCONFIG="/bart/user1/tuxconfig"
*SERVERS
server          SRVID=1
*SERVICES
AVERAGE
```

Figure A1.5: A sample Tuxedo configuration file (UBBconfig).

Notice that the location of the executable file for "server" is specified and the service "AVERAGE" provided by server is specified. The UBBconfig is an ascii file. Before booting the system, the file needs to be compiled into a more efficient form for Tuxedo to read while the Tuxedo system is running. The compiled file is named "tuxconfig." Here is an example of how the UBBconfig file could be compiled:

tmloadcf UBBconfig -y

The "-y" option says to go ahead and overwrite an old copy of tuxconfig, if it exists. We are now ready to boot the Tuxedo system and run the client.

A1.2.4 Booting Tuxedo and Running the Client

We boot the Tuxedo system with the following command:

tmboot -y

The "-y" here says to boot all servers specified in the UBBconfig file. Messages will appear as servers are booted. If errors occur, we will be notified. When the boot process is complete, Tuxedo is running and the server is available.

We can now enter the following command from the command line in another window:

average 1 10 7

and the response

Average (1, 10, 7) = 6

will appear. We can run the average program more times with different data to test the application. When we are done running the system, we enter

tmshutdown -y

which brings down "server" and shuts down Tuxedo. The "-y" here tells Tuxedo you want it to shut down all the servers and go away.

Appendix II. The Tuxedo Configuration File

This section is an overview of the Tuxedo configuration file (UBBconfig). It is important in the rest of this document to understand the role the configuration file plays in the development of applications. The section is meant to show the major capabilities of the configuration file and the sections which comprise the file. It is not a comprehensive list of the capabilities of the configuration file, which are available in the man pages. All the sections of the configuration file will be discussed in Section II, except the ROUTING section, which will be discussed in Section III. Many of the subjects in Section III will involve setting parameters in the UBBconfig file.

A2.1 RESOURCES

The RESOURCES section of the UBBconfig file specifies information about the account Tuxedo will run in and basic information about memory, clients, and servers. The following are the basic parameters specified in the resources section:

- IPCKEY - Well-known address
- MASTER - Master Processor
- UID - UNIX User Identifier of the System/T Administrator
- GID - UNIX Group Identifier of the System/T Administrator
- PERM - Permissions of Access to IPC Structures
- MAXACCESSERS - Maximum number of assessors which can run on any site
- MAXSERVERS - Maximum number of servers which can run on all sites combined
- MODEL - Single Host Machine (SHM), Multiple Processors (MP)
- LDBAL - Should Tuxedo load-balance service requests.

The IPCKEY is the well known address for shared memory (if used). When shared memory is allocated, a unique key must be assigned by the application as a handle to that memory. Tuxedo will use the value placed in this parameter of the resources section.

The MASTER is the initial master site of the Tuxedo system. Tuxedo will be booted from the initial master site. If the application is running on only one machine, that is the master site. If an application spans multiple sites, one site must be designated as the master. The master site can change during run time. The name of the site designated as the master must be defined in the MACHINES section of the UBBconfig file. The UID and GID are the user and group numbers of the System/T administrator. In other words, the account where Tuxedo will run.

The PERM field refers to the permissions which will be assigned to the IPC structures. When shared memory is allocated, permissions are assigned as to who is allowed to do what to the memory (just like files). For example, permissions 0640 means that the owner of the IPC structures can read and write the memory, members of the owners group can read the memory, and the rest of the world cannot read or write the memory. The suggested value for this field is 0660, which should be used unless the application programmer has a specific reason for changing that value.

The MAXACCESSERS and MAXSERVERS specify the number of clients and servers, respectfully, that the system can support. One difference between the how the MAXACCESSERS and MAXSERVERS

parameters are used is that MAXACCESSERS refers to the maximum number of clients supportable on each site in the system, while MAXSERVERS refers to the maximum number of servers supportable in the entire system. These values cause static tables to be allocated, which means they cannot be increased at run-time.

The MODEL tells Tuxedo whether the application will run on a single processor (Single Host Machine, SHM) or on multiple processors (MP). Tuxedo refers to systems running on multiple processors as propagated systems. Propagated systems can be thought of as synonymous with distributed systems.

The LDBAL flag tells Tuxedo whether you want it to load balance service requests. If LDBAL is set to "Y," then Tuxedo will load balance. Service requests are assigned "load" values based on information such as the amount of processing that is required to perform the service and the type of machine the server is running on. For example, a long database query could be assigned a load of 100 while a service which does in memory calculations could be assigned a load of 25. If the same service were provided on two machines where one is twice as fast as the other, the service provided by the slower machine could be assigned a load value twice the larger machine, forcing twice as many requests to the machine which can process them twice as fast. By default, all services are assigned the same load value, and different load values must me determined and changed explicitly by the programmer.

If servers 1 and 2 provide service A, when a service request for service A is made, Tuxedo will determine whether the sum of the load values of the service requests in the queue of server 1 or server 2 is smaller and place the request in that queue. If LDBAL is set to "N," then Tuxedo will round robin service requests between the servers which provide the service. Here is an example of the RESOURCES section for a sample application:

```
*RESOURCES
IPCKEY          100101
MASTER          SITE1
UID             201
GID             20
MAXACCESSERS    50
MAXSERVERS      15
PERM            0660
MODEL           SHM
LDBAL           Y
```

Figure A2.1: A sample RESOURCES Section of the UBBconfig file.

A2.2 MACHINES

The second section of the UBBconfig file is the MACHINES section. The MACHINES section defines logical machines. A logical machine is a machine which Tuxedo recognizes as a separate site on the network. Logical machines are differentiated by physical machines, which involve a separate box with a CPU. Multiple logical machines can appear on a single physical machine in a Tuxedo system, but a logical machine cannot span multiple physical machines. The following parameters are used to define a logical machine in the MACHINES section:

- LMID - Logical Machine ID (a name to reference the logical machine in other sections)
- TUXCONFIG - Full path name of the tuxconfig file for the logical machine

- ROOTDIR - Full path name of the location of the Tuxedo software (binaries, etc.)
- APPDIR - Directory where the application software (servers) is located.

Multiple logical machines can be defined on a single processor simply by specifying multiple definitions in the MACHINES section. Following is a sample MACHINES section of a UBBconfig file:

```
*RESOURCES
IPCKEY          100101
MASTER          SITE1
UID             201
GID             20
MAXACCESSERS    50
MAXSERVERS      15
PERM            0660
MODEL           SHM
LDBAL           Y
*MACHINES
bart            LMID=SITE1
                TUXCONFIG="/bart/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/bart/users/mortgage"
lisa            LMID=SITE2
                TUXCONFIG="/lisa/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/lisa/users/mortgage"
```

Figure A2.2: A sample MACHINES Section of the UBBconfig file.

A2.3 GROUPS, SERVERS, and SERVICES

Three sections which are very related and will be discussed together are the GROUPS, SERVERS, and SERVICES sections. As we discussed before, services are provided by servers. A single service could be provided by one or more servers. A server can provide one or more services. Groups are collections of servers. A server must belong to one, and only one, group. All the servers in a system can belong to the same group. Similarly, groups must run on one, and only one, logical machine. Therefore, the following relationship must hold:

Servers $\subseteq$ Groups $\subseteq$ Logical Machines $\subseteq$ Physical Machines.

There are two main purposes for groups. The first is that groups are used to define where servers run in propagated systems. The second is for data dependent routing. We will look at an example of a propagated system shortly and an example of data dependent routing in Section III. Here are the main parameters for group definitions:

- Group Name - Alphanumeric name of group for reference in other sections
- GRPNO - Unique numeric identifier (within the system) for group
- LMID - LMID of the logical machine where the group is to be located.

Here are the main parameters of server definitions:

- Server Name - Name of the executable file
- SRVGRP - Name of the group this server belongs to
- SRVID - Unique numeric identifier (within the group) for server.

All services provided by servers in the system are listed in the SERVICES section. The service names do not require any parameters. Notice that the group identifiers must be unique within the entire system, while server identifiers need only be unique within the group the server belongs to. This is because group definitions are used to propagate servers across the network. A group can be located anywhere on the network while a server is only referenced within its group.

Suppose we wrote an application with two services, "MORTGAGE" and "VALIDATE." MORTGAGE calculates the monthly mortgage payment that would be required for a loan based on information such as the number of years of the loan, the amount of the loan, and the interest rate. VALIDATE is a service which is called from MORTGAGE to validate the parameters (e.g. the interest rate is greater than 0). We will now look at a possible configuration for the GROUPS, SERVERS, and SERVICES sections of the UBBconfig file for the system. The MORTGAGE service will be provided by an executable named "mortgage" and the VALIDATE service will be provided by an executable named "validate." The system will also be propagated across the bart and lisa logical machines defined in the MACHINES section.

```
*RESOURCES
IPCKEY          100101
MASTER          SITE1
UID             201
GID             20
MAXACCESSERS    50
MAXSERVERS      15
PERM            0660
MODEL           SHM
LDBAL           Y
*MACHINES
bart            LMID=SITE1
                TUXCONFIG="/bart/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/bart/users/mortgage"
lisa            LMID=SITE2
                TUXCONFIG="/lisa/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/lisa/users/mortgage"
*GROUPS
DEFAULT:        LMID=SITE1
MORTGAGE1       GRPNO=1
MORTGAGE2       LMID=SITE2 GRPNO=2
VALIDATE        LMID=SITE2 GRPNO=3
*SERVERS
DEFAULT:        SRVGRP=MORTGAGE1
calculate       SRVID=1 MIN=3
DEFAULT:        SRVGRP=MORTGAGE2
calculate       SRVID=1
validate        SRVID=2 MIN=3 MAX=6
*SERVICES
MORTGAGE
VALIDATE
```

Figure A2.3: Sample GROUPS, SERVERS, and SERVICES Sections of the UBBconfig file.

The UBBconfig shown demonstrates several concepts regarding GROUPS and SERVERS that are worth discussing at this point. The "DEFAULT" statement works the same way in both sections. Parameters specified as DEFAULT are just that, the default for subsequent statements. For example, in the GROUPS section, "LMID=SITE1" is specified as the default. Since the MORTGAGE1 group definition has no LMID specified, it can assumed to be on SITE1. Since the LMID is specified in MORTGAGE2 and VALIDATE, the DEFAULT is not used. The SERVERS section demonstrates that defaults can change. Here, the DEFAULT is set to the SRVGRP that the servers are in and changed the servers for different groups are defined.

The other point of interest from the example are the MIN and MAX statements. The first calculate declares a MIN of 3. This means that we are actually declaring that three copies of calculate will be started in the group MORTGAGE1. In the second calculate declaration, there is no MIN statement, which means only one copy of the calculate server is in group MORTGAGE2. The validate server, however, declares a MIN of 3 and a MAX of 6. This means that three servers will actually be started, but

there is a capability of running up to 6 copies of validate. That means that up to three more copies of validate could be started at run time. The MAX defaults to the MIN, which means that no more copies of calculate could be started at run time, since the maximum number of copies of each are already running. The only way to increase the MAX is to shutdown Tuxedo and change the values.

A2.4 NETWORK

The NETWORK section describes the underlying network in propagated systems. Each site in the network must be defined. Here are the main attributes of an entry in the NETWORK section:

- NADDR - the network address
- BRIDGE - the device of the provider
- NLSADDR - the network listener address.

The NADDR is the network address Tuxedo is to use for the site. For example, if a site is on a UNIX based machine on an ethernet LAN, the network address would be a combination of the hardware address of the ethernet card, the IP address the machine is using, and the port Tuxedo is to use. Every ethernet card has a unique address which is a combination of the manufacturer and a unique number assigned by the manufacturer for its cards. The NLSADDR uses the same format as the NADDR, except the information is that which the Tuxedo listener is to use. The Tuxedo listener is a demon which you run on all sites except the master site. When you boot Tuxedo on the master site, the UBBconfig file is distributed to the other sites through the listener. Tuxedo must know both where the listener is listening and where Tuxedo is to run once the connection is established.

The BRIDGE is the device of the provider. For example, a UNIX based machine would likely be using tcp/ip (the tcp device driver). An IBM site, on the other hand, may be using LU6.2. Here is an example of a NETWORK section:

```
*RESOURCES
IPCKEY          100101
MASTER          SITE1
UID             201
GID             20
MAXACCESSERS    50
MAXSERVERS      15
PERM            0660
MODEL           SHM
LDBAL           Y
*MACHINES
bart            LMID=SITE1
                TUXCONFIG="/bart/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/bart/users/mortgage"
lisa            LMID=SITE2
                TUXCONFIG="/lisa/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/lisa/users/mortgage"
*GROUPS
DEFAULT:        LMID=SITE1
MORTGAGE1       GRPNO=1
MORTGAGE2       LMID=SITE2 GRPNO=2
VALIDATE        LMID=SITE2 GRPNO=3
*SERVERS
DEFAULT:        SRVGRP=MORTGAGE1
calculate       SRVID=1 MIN=3
DEFAULT:        SRVGRP=MORTGAGE2
calculate       SRVID=1
validate        SRVID=2 MIN=3 MAX=6
*SERVICES
MORTGAGE
VALIDATE
*NETWORK
SITE1
                NADDR="0x000223CC031305E6"
                BRIDGE="/dev/tcp"
                NADDR="0x000223CC031305E6"
SITE2
                NADDR="0x000223CC031305D9"
                BRIDGE="/dev/tcp"
                NADDR="0x000223CC031305D9"
```

Figure A2.4: A sample NETWORK Section of the UBBconfig file.

It would be intuitively obvious to the most casual observer that the 16 digit hexadecimal numbers for the NADDR and NLSADDR are the same on each site. This only means that Tuxedo is going to run on the same address that the listener was using, which is perfectly acceptable.

Appendix III. Tuxedo Clients and Servers

This section discusses how to develop clients and servers. The section provides descriptions and sample code for writing clients and servers, making synchronous and asynchronous service requests, and how to do unsolicited server to client communications.

A3.1 Tuxedo Clients and Servers

We will now discuss writing Tuxedo clients and servers.

A3.1.1 Tuxedo Clients

There are four major issues regarding writing Tuxedo clients: initializing with Tuxedo, memory allocation and deallocation, making service calls, and terminating the connection to the Tuxedo Transaction Monitor (TM).

Before a client can make service requests, it must call tpinit( ) to initialize the connection with the TM. The parameter to tpinit( ) is a pointer to a TPINIT structure. The client can use the TPINIT structure to give information to Tuxedo about the client. For example, the client can say whether Tuxedo is allowed use signals for communications in the client process. The client can also use the structure to name itself, which enables other processes to send information to the client using its name. The client can also pass a NULL pointer to a TPINIT structure to tpinit( ). Here is an example of a call to tpinit( ):

```
if ( tpinit( (TPINIT *)NULL ) == -1 )
{
    printf( "tpinit( ) failed \n" );
}
```

The next topic we will discuss is memory allocation and deallocation. The function tpalloc( ) allocates a block of memory from the heap, tprealloc( ) reallocates an allocated block to be a different size, and tpfree( ) returns the memory to the heap. Clients must allocate memory for the message buffers in order to make service requests with these calls instead of malloc( ) and free( ). When Tuxedo allocates memory, through tpalloc( ), it keeps internal structures describing the amount and type of data it contains. Tuxedo also may reallocate the memory during service requests. If a message buffer allocated by malloc( ) is used for a service request, the request will fail. Here is an example of tpalloc( ) and tpfree( ):

```
char *buf;

if (( buf = tpalloc( "CARRAY", NULL, 100 )) == -1 )
{
    printf( "tpalloc( ) failed\n" );
}

tpfree( buf );
```

The "CARRAY" parameter of tpalloc( ) refers to the buffer type. CARRAY means the memory is binary data. CARRAY and the buffer types supported by Tuxedo will be discussed further later in this document. The parameter 100 refers to the size of the buffer to be allocated.

We will now discuss how to make synchronous service calls. Service calls will be discussed more fully in the next section. A synchronous service call means that when we return from the service request, we already have the reply from the server. The function to make synchronous service requests is tpcall( ). Here is an example of a call to tpcall( ):

```
char *buf;
int len = 100;

if (( buf = tpalloc( "CARRAY", NULL, len )) == -1 )
{
    printf( "tpalloc( ) failed \n" );
}

/* fill in buf */

if ( tpcall( "SERVICE", buf, len, &buf, &len, 0 ) == -1 )
{
    printf( "tpcall( ) failed \n" );
}
```

It the tpcall( ) function, the "SERVICE" parameter is the name of the service being requested. Notice that in this example, buf (the allocated buffer) and len (the length of buf) are passed in. The &buf and &len are then passed in to retrieve the return message. The final parameter is a flags field we can use to give Tuxedo information about how to process the request.

The other basic function used by clients is tpterm( ), which tells the TM we are through making Tuxedo service calls. Normally, tpterm( ) is called when the client is about to exit( ), but that is not required. Here is an example of a tpterm( ) call:

```
tpterm( );
exit( 0 );
```

A3.1.2 Tuxedo Servers

Tuxedo servers provide services. By default, the service has the same name as the function in the server which processes the service request. We will assume that the service and function names are the same, and if the reader wishes to use different function names than service names, they will refer to the man pages to determine how to do that. We will now look at how the function which processes a service request is written. We will then look at how initialization and termination in servers can be performed and how to pass command line arguments to a server.

A3.1.2.1 Processing Service Requests

When a server receives a service request, the function which knows how to provide the service is executed by Tuxedo. The parameter to the function providing the service request is a handle to a TPSVCINFO structure. The TPSVCINFO structure has fields, including: "data" (the message sent by the application), "len" (the size of "data"), and "cltid" (a Tuxedo identifier for the client process). The server returns a response to the client making the service request by calling a Tuxedo function called tpreturn( ). The tpreturn( ) function includes a flag stating whether the service request succeeded or failed. Here is an example of a function which processes a service called "MYSERV":

```
MYSERV( TPSVCINFO *msg )
{
    < code here to process service >

    /* return result of service request */
    if ( < error occurred > )
    {
        tpreturn( TPFAIL, 0L, NULL, 0, 0L );
    }
    else
    {
        tpreturn( TPSUCCESS, 0L, msg->data, msg->len, 0L );
    }
}
```

The MYSERV function shows how a service can return to the application whether the service request was successfully completed or not (TPFAIL or TPSUCCESS respectively). When the service request completes successfully, MYSERV returns a message in the same buffer which it is given (msg->data). The service could have allocated a new buffer (with tpalloc( )) and returned that buffer. The service also could have returned part of the buffer passed in to the function (i.e. the buffer msg->data, but a length less than msg->len). The message could not have used msg->data with a length greater than msg->len since msg->len tells the service how large the buffer is.

A3.1.2.2 Initialization, Termination, and Command Line Options (CLOPT)

Writing servers is not the same as writing clients since servers do not have a main. Servers, however, may need to do initialization and termination. For example, the server may need to initialize global variables, allocate memory, and/or do database processing when it is booted. The server may also need to deallocate memory and do other database processing when the server shuts down. Also, client processes can receive command line arguments through in the main. Command line arguments may be needed in servers too, but servers have no main. We will now discuss how to do initialization and termination in servers and how to pass command line arguments to servers.

Initialization and termination are done by binding the functions tpsvrinit( ) and tpsvrdone( ) respectively into servers. Either both or neither functions must be bound in. In other words, you cannot bind in tpsvrinit( ) without binding in tpsvrdone( ). These functions should not be confused with tpinit( ) and tpdone( ). The functions tpinit( ) and tpdone( ) are for clients, and tpsvrinit( ) and tpsvrdone( ) are for servers. Also, tpinit( ) and tpdone( ) are functions which are called by clients, and tpsvrinit( ) and

tpsvrdone( ) are simply bound into servers and are called by Tuxedo to perform initialization and termination.

As previously stated, tpsvrinit( ) is called by Tuxedo when the server is booted. Servers can be booted at startup time or at any time during the execution of the Tuxedo system. Any time a server is booted, tpsvrinit( ) is called, and any time it shuts down, tpsvrdone( ) is called, so the functions are not only called when Tuxedo itself is booted or shuts down. If an application error occurs in tpsvrinit( ) (e.g. out of memory), tpsvrinit( ) can return an error code. The error will be logged and Tuxedo will know that the server did not start up properly.

Command line arguments for servers are specified in the UBBconfig file in the server section. The Command Line Option (CLOPT) parameter is used. We will now look at an example of the format of the CLOPT command and what it means:

CLOPT="-A -- -f firstparam -s secondparam"

The format of the statement is that options before the "--" are for Tuxedo and options after the "--" are for the application. A complete list of Tuxedo options are in the man pages. An example of a Tuxedo option is -A, which means that all services build into the server are advertised. An advertised service means that the server which advertised it is available to receive service requests for that service. Tuxedo will not give service requests to servers which have the capability of providing a service, but have not advertised the service. Applications can advertise and unadvertise services for specific servers at any time that they choose for whatever reason they choose. By default, all services built into a server are advertised.

Another example of a Tuxedo option in CLOPT is "-o filename." The -o option instructs Tuxedo to redirect stdout in the Server into the file named "filename." The default CLOPT is CLOPT="-A --", which means that all services built into the server are advertised, and there are no application options. Here is an example of a UBBconfig file with a CLOPT:

```
*RESOURCES
IPCKEY          100101
MASTER          SITE1
UID             201
GID             20
MAXACCESSERS    50
MAXSERVERS      15
PERM            0660
MODEL           SHM
LDBAL           Y
*MACHINES
bart            LMID=SITE1
                TUXCONFIG="/bart/users/mortgage/tuxconfig"
                ROOTDIR="/tux"
                APPDIR="/bart/users/mortgage"
*GROUPS
MORTGAGE        LMID=SITE1 GRPNO=1
*SERVERS
DEFAULT         SRVGRP=MORTGAGE
calculate       SRVID=1 CLOPT="-A -- -f firstparam -s secondparam"
validate        SRVID=2
*SERVICES
MORTGAGE
VALIDATE
```

Figure A3.1: An example of a UBBconfig with Command Line Options (CLOPT).

The example shows that each server can have a different CLOPT specification. That means that different copies of the same server can be given different parameters to customize processing in each server. One limitation to the CLOPT is that the UBBconfig cannot be passed parameters which are in turn passed through a CLOPT to a server. For example, shell scripts in the Korn shell can be passed arguments which can be referenced in the script as $1 for argument 1, $2 for argument 2, etc. If dynamic parameters need to be passed to servers, environment variables can be set in before the server is invoked and the server can call getenv( ) to retrieve the information.

In our example, we have passed command line options to the server. We now need to know how to retrieve the information in the server. The application's command line options are passed into tpsvrinit( ), with the same format as they would be passed into the main in a client (i.e. int argc and char *argv[ ]). The server can then use the standard "C" library routing getopt( ) to retrieve parameters. Here are examples of tpsvrinit( ) and tpsvrdone( ) using the given example of the UBBconfig file:

```
int tpsvrinit( int argc, char *argv[ ] )
{
    int c;
    extern char *optarg;
    extern int optind;

    userlog( "Server is booting" );
    while (( c = getopt( argc, argv, "f:s:" )) != EOF )
    {
        switch( c )
        {
            case 'f':
                userlog( "The first parameter is %s", optarg );
                break;
            case 's':
                userlog( "The second parameter is %s", optarg );
        }
    }
    return( 0 );
}

void tpsvrdone( void )
{
    userlog( "Server is shutting down" );
    return( );
}
```

Figure A3.2: An example of tpsvrinit( ), tpsvrdone( ), and CLOPT in a server.

A3.2 Synchronous and Asynchronous Service Calls

As discussed in earlier sections, Tuxedo supports both synchronous and asynchronous service calls. Synchronous service calls are made using the function tpcall( ). When the client returns from tpcall( ), the reply from the server for the service request is already in a buffer for the client to process. In an asynchronous service call, Tuxedo sends the service request, but does not wait for a reply. The function used to make an asynchronous service request is tpacall( ). When the client calls tpacall( ), a handle related to the service request is returned to the client. If the client is not interested in the reply, it can pass a "NOREPLY" flag into tpacall( ), and Tuxedo will discard the reply; otherwise, the client calls tpgetreply( ) later to retrieve the reply.

If multiple service requests have been made, the function tpgetreply( ) can be called to look for a reply from a specific service request, or it can be called to look for a reply from any service request that has sent a reply. To have tpgetreply( ) only look for one reply, the client can pass the handle returned by tpacall( ) when the service request was made. To have tpgetreply( ) return any reply that comes in, the flag "GETANY" is passed in replace of a handle. When tpgetreply( ) returns a reply, it also returns the handle for the service request which was returned by tpacall( ) when the request was made. The client can then compare that handle to its outstanding service request handles to determine which service request the reply is related to.

The tpgetreply( ) function can also be called in wait or no wait mode. In wait mode, tpgetreply( ) does not return until a reply is received. In no wait mode, tpgetreply( ) returns immediately, whether there is a reply or not. The client can use tpgetreply( ) wait mode to make parallel service requests. For example, the client can call services A, B, C, and D using tpacall( ), and then wait for the replies together, rather than calling A, waiting for a reply from A, calling B, waiting for a reply from B, etc. Using tpgetreply( ) no wait mode, the client can do other processing and periodically check for replies. We will now look at an example of tpacall( ) making two service requests in parallel:

```
char    *buf1, *buf2
int     len1 = 20, len2 = 20;
int     han1, han2;

/* allocate a buffer for the service request */
if   ((( buf1 = tpalloc( "CARRAY", NULL, len1 )) <= 0 ) ||
      (( buf2 = tpalloc( "CARRAY", NULL, len2 )) <= 0 ))
{
    < process error:  could not allocate memory >
}

< fill in buf1 and buf2 >
if   (( han1 = tpacall( "SERVICE1", buf1, len1, 0 ) == -1 ) ||
      (( han2 = tpacall( "SERVICE2", buf2, len2, 0 ) == -1 ) ||
{
    < process error:  was not able to send a service request >
}

< do other work >

while ( < requests both not yet returned > )
{
    handle = tpgetreply( GETANY, &buf1, &len, 0 );
    if ( handle = han1 )
    {
        < process reply from "SERVICE1" >
    }
    else if ( handle = han2 )
    {
        < process reply from "SERVICE2" >
    }
}
```

Figure A3.3: An example of tpacall( ).

The server does not care whether the service request is made in synchronous or asynchronous mode. The only difference is how and when the client retrieves the reply. It is up to the application programmer to determine what calling method should be used. For example, if the client wants to request services A and B, and the client needs data from the reply from service request A for service request B, then tpcall( ) should be used to request service A.

A3.3 Unsolicited Server to Client Communications

We have looked at service requests and how servers generate "replies" to those requests. Replies are solicited server to client communications since the client explicitly requested the service knowing that a reply would be generated. What if a service request from one client generates a need for communications to another client, multiple clients, or even all clients in the system. The client making the service request is soliciting a reply, but the other clients would be receiving unsolicited messages since they had not requested any services. There are two forms of unsolicited messages in Tuxedo: unsolicited messages to specific clients and unsolicited messages to groups of clients (or all clients).

For an example of why we might like to send unsolicited messages to clients, let us consider a stock market application with investors and traders. Each investor and trader has a client which makes service requests when needed. When an investor wants to order stock from a trader, the client makes a service request to a server, which at some point will cause an unsolicited message to the trader's client indicating that the trader has an order from the investor. The message to the trader's client is unsolicited since it asked for no services to be performed.

Suppose an investor's regular trader is not on line. The system might generate a message to all traders working in the same brokerage house as the investors regular trader asking if any of them would be willing to fill the order. This is an example of a message to a group of clients, rather than a specific client. If the system was about to come down for maintenance, a message could be generated to all clients indicating that they have two minutes to complete their work and log off. Tuxedo supports the ability to send unsolicited messages to specific clients, groups of clients, and all clients.

We will first discuss how servers send unsolicited messages to specific clients. We will then discuss how clients can receive unsolicited messages. There are two ways the client can retrieve unsolicited messages: by polling or through signals. Finally, we will discuss sending unsolicited messages to groups of clients and all clients.

A3.3.1 Unsolicited Messages to Specific Clients (tpnotify( ))

The function used to send a message to a single client is tpnotify( ). The tpnotify( ) function takes a Tuxedo client identifier (called cltid) as a parameter indicating the client who is to receive the message. The cltid of the client making a service request is passed into the service as a field of the TPSVCINFO structure, which is the parameter for servers when they receive service requests. Every client in the system has a unique Tuxedo client identifier. If a client makes a service request to a server, the server can store the client id from the client, and use the client identifier to send unsolicited messages to the client.

For example, in the stock system, when traders are available to receive orders, they could make a service request to the order system in our stock marked example informing the order system that they are available to receive orders. When investors submit orders to the order system, the order system determines which trader is to receive the order and sends an unsolicited message to the trader's client to buy 100 shares of General Electric. Here is an example of a server in the order system sending an unsolicited message to the trader's client using tpnotify( ):

```
ORDER( TPSVCINFO *msg )
{
    CLIENTID *trader;

    < code here to process order >

    if ( tpnotify( <client id of trader>, msg->data, msg->len ) != -1 )
    {
        < trader was notified of order >
    }

    /* return result of service request */
    if ( < error occurred > )
    {
        tpreturn( TPFAIL, 0L, NULL, 0, 0L );
    }
    else
    {
        tpreturn( TPSUCCESS, 0L, msg->data, msg->len, 0L );
    }
}
```

Figure A3.4: An example of a server calling tpnotify( ).

A3.3.2 The Unsolicited Message Handler

Now that the server has sent an unsolicited message to a client, how does the client receive the message? To receive the message, the client provides a function called the unsolicited message handler. The unsolicited message handler is executed when the client receives a message. Therefore, the handler must be reentrant, since it does not know what the client process was doing when it was invoked. The processing of the function should be to save the message in a global queue of data so that the message can be processed later at a safe time. We will discuss how to write an unsolicited message handler, how to have Tuxedo dispatch the handler to retrieve unsolicited messages, and how to write the event loop of the client to process the message safely.

We will first discuss the format of the unsolicited message handler. There are five Tuxedo functions which are allowed to be called from the event handler: tpalloc( ), tprealloc( ), tpfree( ), tpgetlev( ), and tptypes( ). We have already discussed tpalloc( ), tprealloc( ), and tpfree( ). The tpgetlev( ) function tells the handler whether or not the system is in transaction mode (i.e. global transactions are enabled). The function tptypes( ) tells the unsolicited message handler the type of buffer. Because of the limited number of Tuxedo calls allowed, the handler cannot do significant processing. Reentrant "C" functions are also allowed in the unsolicited message handler. Here is an example of an unsolicited message handler:

```
ProcessUnsol( char *data, long len, long flags )
{
    char *buf;

    if (( buf = tpalloc( "CARRAY", NULL, len )) != -1 )
    {
        memcpy( buf, data, len );
    }

    < add buf to a global queue of data >

    return;
}
```

Figure A3.5: An example of an unsolicited message handler.

Notice that all the calls in the handler are one of the listed "safe" Tuxedo calls and reentrant "C" functions. Notice also that the message is saved and not processed. The next example shows the main program, which declares the unsolicited message handler and processes unsolicited messages. In the example, we will use polling to look for unsolicited messages. We will then look at how to dispatch the unsolicited message handler automatically using signals.

```
main( int argc, char *argv[ ] )
{
    /* tell Tuxedo the name of the unsolicited message handler */
    tpsetunsol( ProcessUnsol );

    while( 1 )
    {
        /* poll for unsolicited messages */
        tpchkunsol( );

        /* process unsolicited messages */
        while( < global queue of data > != < empty > )
        {
            < remove item from queue and process >
        }

        sleep( 1 );
    }

    exit( 0 );
}
```

Figure A3.6: Dispatching an unsolicited message handler using polling.

The function tpsetunsol( ) is called once at the beginning of the program to inform Tuxedo what function the client would like to use to process unsolicited messages. Any Tuxedo call may dispatch the message handler if a function is received. The function tpchkunsol( ) is a no-op (no operation) Tuxedo call which dispatches all unsolicited messages which have been received. We call that function before checking the

queue to process messages to be sure that all messages received are in the queue. Remember, our unsolicited message handling function only put the messages in a queue.

Next, we process all the messages from the queue. Processing the messages in an event loop is safe since we know we are not interrupting processing. The processing here does not need to be reentrant. One of the drawbacks to the polling method is the sleep(1), which means that we are waking up every second to determine if we have a message. Most of the time we will probably not have a message. The solution to the polling problem is the use of signals. We will now look at the same code using signals.

The use of signals is declared in the configuration file. In the resources section, the following parameters can be declared:

```
*RESOURCES
IPCKEY          100101
MASTER          SITE1
UID             201
GID             20
MAXACCESSERS    50
MAXSERVERS      15
PERM            0660
MODEL           SHM
LDBAL           Y
NOTIFY          SIGNALS
SIGNAL          SIGUSR2
```

Figure A3.7: UBBconfig for unsolicited message handling using signals.

The NOTIFY parameter tells Tuxedo that the application wishes to use signals for unsolicited message handling. The SIGNAL parameter tells Tuxedo what signal to use for notification. The options are SIGUSR1 and SIGUSR2. The SIGNAL parameter is optional, and defaults to SIGUSR2. Here is how the main program is modified when using signals.

```
main( int argc, char *argv[ ] )
{
    /* tell Tuxedo the name of the unsolicited message handler */
    tpsetunsol( ProcessUnsol );

    while( 1 )
    {
        /* process unsolicited messages */
        while( < global queue of data > != < empty > )
        {
            DEFERSIGS( );
            < remove item from queue >
            RESUMESIGS( );
            < process item removed from queue >
        }

        sleep( 999999 );
    }

    exit( 0 );
}
```

Figure A3.8: Dispatching an unsolicited message handler using signals.

Notice that the main program is basically the same. We still call tpsetunsol( ) to tell Tuxedo the name of the unsolicited message handler. The first difference is that we do not need to call tpchkunsol( ) to dispatch the handler for any received messages which have not yet been processed. This is because the handler was dispatched by a signal as soon as the message was received. The next difference is that we must defer signals when we are removing items from the queue, since if another message came in at that point the handler could be adding an item to the queue while we are removing it here. That must not be allowed to happen. The other difference is that our sleep time is now large since we do not need to continually wake up to check for unsolicited messages. The sleep( ) function automatically wakes up when a signal is received. This triggers our checking the queue and processing the message.

A3.3.3 Unsolicited Messages to Groups of Clients (tpbroadcast( ))

The function which can send unsolicited messages to groups of clients or all clients is called tpbroadcast( ). Just as the server does not know how the client is processing messages (polling or signals), the client does not know how the server is sending unsolicited messages (tpnotify( ) or tpbroadcast( )). The client code is the same. The only difference in the functionality of tpbroadcast( ) as opposed to tpnotify( ) is that tpbroadcast( ) can be used to address multiple clients or all clients with one function call.

To address a group of clients, the clients must name themselves in tpinit( ). Here is an example (from the stock example) of trader clients naming themselves traders:

```
#include <stdio.h>
#include <atmi.h>

int main( int argc, char *argv[ ] )
{
    TPINIT *tpinfo;

    if (( tpinfo = tpalloc( "TPINIT", (char *)NULL, sizeof( TPINIT ))) == NULL )
    {
        < process error >
    }

    /* name the client "trader" */
    strcpy( tpinfo->cltname, "trader" );

    /* connect to tuxedo */
    if ( tpinit( tpinfo ) != -1 )
    {
        . . .

        tpterm( );
    }

    exit( 1 );
}
```

Figure A3.9: A Tuxedo client naming itself.

A TPINFO structure allows the client to specify certain startup parameters, including a client name. Other parameters are referenced in the man pages. Notice we pass the tpinfo structure into the call to tpinit( ). We will now look at how a server in the order system could broadcast to all the traders using the name from the last example:

```
ORDER( TPSVCINFO *msg )
{
    < process the order, determine the investors trader is not on line >

    if ( tpbroadcast( (char *)NULL, (char *)NULL, "trader", < data >, < len >, < flags > ) == -1 )
    {
        tpreturn( TPFAIL, 0L, NULL, 0, 0L );
    }
    else
    {
        tpreturn( TPSUCCESS, 0L, < data >, < len >, < flags > );
    }
}
```

Figure A3.10: A Tuxedo server sending an unsolicited message to a group of clients.

In order to broadcast to all clients, the server specifies NULL as the client name. Here is an example of a broadcast call to all clients:

```
ORDER( TPSVCINFO *msg )
{       .
    < process the order, determine the investors trader is not on line >

    if ( tpbroadcast( (char *)NULL, (char *)NULL, (char *)NULL,
                                    < data >, < len >, < flags > ) == -1 )
    {
        tpreturn( TPFAIL, 0L, NULL, 0, 0L );
    }
    else
    {
        tpreturn( TPSUCCESS, 0L, < data >, < len >, < flags > );
    }
}
```

Figure A3.11: A Tuxedo server sending an unsolicited message to all clients.

This ends the discussion of client and server communications. Next, we will look at how we can use the clients and servers to manage global transactions.

References

[1]Leff, Avraham and Pu, Calton, "A Classification of Transaction Processing Systems," Computer, June 1991, Vol. 24 No. 6, pp. 63-76.

[2]X/Open Company Limited, "Distributed Transaction Processing: The XA Specification," 1991.

[3]Independence Technologies, Incorporated, "Open Systems Transaction Processing Market Overview," 1992.

[4]McLachlan, Gordon, "Decision '92: Distributed OLTP," HP Professional, 1992, pp. 26-32.

[5]Mafla, Enrique and Bhargava, Bharat, "Communication Facilities for Distributed Transaction-Processing Systems," Computer, August 1991, Vol. 24, No 8, pp. 61-66.

[6]Mantelman, "The birth of OSI TP: A new way to link OLTP networks," Data Communications, November 1989, Vol. 18, No. 14, pp. 141-148.

[7]Unix® System Laboratories, Incorporated, "TUXEDO® Enterprise Transaction Processing System, Release 4.2 Product Overview," 1992, pp. 1-20.

[8]Unix® System Laboratories, Incorporated, "TUXEDO® Enterprise Transaction Processing System, Release 4.2 Transaction Processing," 1992.

[9]Independence Technologies, Incorporated, "Corporate Background," May, 1992.

[10]Berson, Alex, "Client/Server Architecture," McGraw Hill, 1992, pp. 347-370.

[11]Letson, Russell, "OLTP migrates to PC LANS," Systems Integration, May 1990, Vol 23, No. 5, pp. 40-48.

[12]Livingston, Dennis, "American Express reins in the paper," Systems Integration, May 1990, Vol 23, No. 5, pp. 52-58.

[13]Richman, Dan, "USL To Unveil Spiffed-Up Tuxedo," Open Systems Today, February 15, 1993, p. 3.

[14]Date, C. J., "An Introduction to Database Systems, Volume II," Addison-Wesley Publishing Company, 1985, Chapters 3 & 7.

[15]Datapro, "Transarc Encina Transaction Processing Software," Datapro Reports on UNIX Systems & Software, August 1992, pp. 101-109.

[16]NCR, "Top End Product Overview," 1992.

[17]"IBM: OLTP and UNIX," Advanced Network Computing, October 5/19 1992, pp. 1123-1132.

[18]Gartner Group Industry Service, "CICS for UNIX: An Optimistic Sign From the New IBM," Inside Gartner Group This Week , November 4 1992, Vol 8, No. 44, pp. 1-4.