INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

High Availability Solution for a Transactional Database System

Steluta Radulescu Budrean

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

March 2002

© Steluta Radulescu Budrean, 2002

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68528-4

Canada

ABSTRACT

High Availability Solution for a Transactional Database System

Steluta Radulescu Budrean

In our increasingly wired world, there is stringent need for the IT community to provide uninterrupted services of networks, servers and databases. Considerable efforts, both by industrial and academic communities have been directed to this end. In this project, we examine the requirements for high availability, the measures used to express it, and the approaches used to implement this for databases. The purpose of this project is to present a high availability solution, using off-the-shelf hardware and software components, for fast fallback and restart in a fast changing transaction-based application. The approach uses synchronous replication, which, in case of a failure, is able to resynchronize the databases without shutting down the system.

AKNOWLEDGEMENT

I would like to thank Prof. Dr. Bipin C. Desai for his guidance and advice all along the way. And I would like to take this opportunity to thank my colleagues Jean Millo and Olivier Epicoco from SITA for their help and guidance during this stressful period, and the A.S.D. Department from SITA for their continuous support.

LIST OF FIGURES

FIGURE 1 CLUSTERED SERVERS 12

FIGURE 2 STORAGE AREA NETWORK 14

FIGURE 3 STANDBY DATABASE REPLICATION 18

FIGURE 4 CLUSTERED DATABASE- SHARED DISK ARCHITECTURE [18] 23

FIGURE 5 CLUSTERED DATABASE-SHARED NOTHING ARCHITECTURE [34] 24

FIGURE 6 OVERALL ARCHITECTURE 40

FIGURE 7 SYSTEM ARCHITECTURE 44

FIGURE 8 X-OPEN DTP MODEL 46

FIGURE 9 CLIENT DESIGN 47

FIGURE 10 TUXEDO APPLICATION SERVER -SERVER GROUP 48

FIGURE 11 TUXEDO MULTIPLE SERVER GROUPS 49

FIGURE 12 APPLICATION DESIGN 51

FIGURE 13 LIST OF SERVERS AND SERVICES 56

FIGURE 14 SEQUENCE DIAGRAM FOR NORMAL MODE 57

FIGURE 15 SEQUENCE DIAGRAM FOR DEGRADED MODE 58

FIGURE 16 SEQUENCE DIAGRAM FOR THE RECOVERY PROCEDURE 59

FIGURE 17 SERVICES STATUS AFTER MACHINE FAILURE 61

FIGURE 18 NODE FAILOVER DEMO 66

FIGURE 19 NODE FALLBACK DEMO 67

LIST OF TABLES

TABLE 2-1 HIGH AVAILABILITY MEASUREMENTS 4

TABLE 6-1: TEST OVERVIEW 63

Table of Contents

1. Introduction *1*

2. The need for High Availability in today's market *3*

2.1. Metrics of High Availability 3

2.2. Causes of Downtime 5

2.3. System Recovery 7

3. High Availability Methods *9*

3.1. Hardware HA Solutions 9

3.1.1. Redundant components *10*

3.1.2. Clustered Servers *10*

3.1.3. Storage Area Networks *14*

3.2. Database HA Solutions 15

3.2.1. Standby Database and Distributed Databases *16*

3.2.2. Parallel Processing, Clusters *21*

3.3. Transactional HA Solutions 27

3.3.1. Transaction Processing *27*

3.3.2. Transactional Systems *29*

3.4. HA Methods - Conclusions 31

4. Replication Solution for ensuring HA in an OLTP environment *33*

4.1. Problem Definition 33

4.1.1. Assumptions and Constraints *34*

4.1.2. Detailed Problem Definition *35*

4.2. Problem Approach and Analysis 36

4.2.1 System and Functional Requirements *36*

4.3. Architectural and Functional Constraints 38

5. Proposed Solution and Its Design *40*

5.1. Design Rationale 40

5.1.1 Choice of architecture *40*

5.1.2. Availability *41*

5.1.3. Overall System Cost *41*

5.1.4. Scalability *42*

5.1.5. Manageability *42*

5.2. System Architecture 43

5.3. System Design 45

5.3.1. Dealing with Transactions *45*

5.3.2. Subsystem Design *46*

6. Implementation, Testing and Results *53*

6.1. Application Configuration 53

6.2. Server and Client Applications 56

6.3. Administrative scripts 60

6.4. Experiment and results 63

7. Conclusion and future work *69*

8. References *72*

9. Appendices *76*

A. Appendix A - Configuration File 76

B. Appendix B - Makefile 82

C. Appendix C - Environment Files 86

D. Appendix D - Glossary 88

1. Introduction

A measure of availability is the length of time during which a system can be used for uninterrupted production work. High Availability (HA) is an extension of that duration, perceived as extended functionality of the system, masking certain outages. High availability can be achieved through reliable components and redundancy: the latter allows a backup system to take over when the primary system fails. In a highly available system, unplanned outages do occur, but they are made transparent to the user[1].

There are degrees of transparency for a system, which imply degrees of high availability and there are two major factors that influence it:

- Type of processing

The availability of a system is quantified differently depending on the type of processing done, such as batch processing or real-time. The requirements to ensure availability of a batch processing system, compared to a real-time system are very different and hence a lot harder to achieve in the latter case, due to stringent time constraints.

We shall direct our research towards the transactional systems that represent more of a challenge in the IT industry (i.e. telecommunication industry, Web transactions processing, banking transactions, etc.)

- The cost of a system

In a highly demanding environment in terms of throughput and transactional changes, the cost of a system that insures "continuous" availability could be very high. By endowing the system with redundant components this goal can be achieved

but the cost will definitely increase over reasonable limits.

The problem, which is addressed in this thesis, is the fast synchronization, in case of failures, of two databases supporting a transactional system. By taking the logic of transaction coordination and write-ahead-log, outside of the database, the system relieves the database of all its responsibility except as a repository and provides a recovery solution, which is not dependent on the availability of the databases.

The organization of this thesis is as follows: In chapter 2, we give an overview of high availability and a measure used for expressing it. In the next chapter, we present some industrial approaches used to provide HA. Chapter 4 outlines replication solution for online transactional processing (OLAP). In Chapter 5, we present our proposed solution and outline our design. Chapter 6 gives the implementation details and the results of our experiments using this implementation. The conclusions are given in the final chapter.

2. The need for High Availability in today's market

The solutions for achieving HA are ranging from complete hardware redundancy to software redundancy such as standby databases and replicated databases in case of distributed systems. The challenge in most HA system is to compensate not only for unplanned outages but also for planned outages as well. In the real world the HA solutions are usually a trade-off between the systems needs and the cost economically justified.

If money is no object, then we can over protect our systems against failure by duplicating everything: even here, we would not attain more than 99.999% availability. However, the goal is to have self-recoverable systems in homogenous and also heterogeneous environments.

Mission-critical environments such as telecommunication industry and on-line business applications have a need for HA databases that offer throughput and real-time requirements.

2.1. Metrics of High Availability

To be able to really quantify and identify the availability of a system, the academic world and the industry has defined some metrics that formalize the definition of a system's availability:

- Mean Time to Recover (MTTR)

MTTR represents the necessary time for a system to recover after a failure. The industry and the academic world today concentrate on finding solutions to reduce this time and to make it transparent to the users.

- Mean Time Between Failures (MTBF)

MTBF is mostly computed based on hardware failures, but today's industry has made significant progress in achieving very good MTBF through redundant hardware and software (e.g. clustered nodes etc.)

Another way of expressing availability is referred to as "number of nines". This is expressed as a percentage of uninterrupted service per year and hence the downtime. A system is considered highly available when its availability is 99.9 also called "3 nines" [2]. As we can see from the table below, the aim of a " five-nines" system is to have less than a few minutes downtime per year:

Availability	Downtime
99.9%	525.6 min or 8.76 hrs
99.99%	52.55 min
99.999%	5.25 min

Table 2-1 High Availability Measurements

There is a certain gray area in computing availability, given by the transparency of recovery, which may or may not be taken into account from the user's point of view. An interesting mathematical quantification is given by Clustra database, in a multi-node/clustered architecture[2]. The factors that are taken into consideration for computing availability are as follows:

- Percent of time a single node is unavailable:

$$Punavailable = \frac{(Nrestart*Trestart) + (Nrepair*Trepair) + (Nmnt*Tmnt) + (Nupdate*Tupdatet)}{24*365}$$

Where:

Punavailable is the percentage of time a single node will be unavailable due to failure or maintenance.

Nrestart is the number of restartable node failures per year

Trestart is the time to recover from a restartable node failure

Nrepair is the number of node failures per year requiring repair

Trepair is the time to repair a node

Nmnt is the number of maintenance operations per year

Tmnt is the time a node is down due to maintenance operations

Nupdate is the number of OS updates per year

Tupdate is the time a node is down during an OS update operation

- Likelihood of a node failure (failure intensity)

$$\text{Ifailure} = \frac{\text{Nrestart} + \text{Nrepair} * T}{24 * 365}$$

Where:

Ifailure is the likelihood of node failure

Nrestart is the number of reastartable node failures per year

Nrepair is the number of node failure per year requiring repair

T is the accelerator (increased likelihood of a second node failure if first one fails) ~ 2

- Hence MTBF can be calculated as follows:

$$\text{MTBF} = \frac{1}{\text{Punavailable} * \text{Ifailure} * \text{Nnodes}}$$

2.2. Causes of Downtime

Downtime is the time interval when the application/database is not available to the users. There are two categories of downtime: planned and unplanned. Both categories have a decisive impact on the availability of the system.

- Planned Downtime
  - ◆ Hardware Upgrades: Most industrial scale databases reside on symmetrical

multiprocessor (SMP) computing systems. While they allow hot upgrades (processors, disks) when the limit is reached, a downtime is required to replace the hardware.

- ◆ Database Software Upgrades: Most databases require the entire database to be shut down when a different version is installed or maintenance release is applied.
- ◆ Operating Systems Upgrades: For the upgrades of the OS the whole systems is shutdown including the database.
- ◆ Application Modifications: When upgrading the application, which also entails database structure modifications (add/modify tables), the application needs to be restarted.

- Unplanned Downtime
  - ◆ Hardware Problems: CPU, Memory, and System failures. Most hardware today is very reliable, however failure may occur, especially if the hardware is not protected through backup components etc.
  - ◆ Software Problems: Operating system, Database. The bugs in OS or database software, are usually few but are hard to detect and difficult to repair. Usually software maintenance releases must be installed, which means again restart of the database system.
  - ◆ Network Problems: Depending on the architecture, network problems may affect the database systems. For example distributed systems can be affected by network failures, since data is periodically copied at different sites.
  - ◆ Human errors: Manual intervention can always introduce the risk of failure.

All the causes identified above contribute to the failure of a system; however in this thesis we will concentrate on the database systems recovery techniques.

2.3. System Recovery

A system that deals automatically with failures passes through two stages: failover and fallback. When a failure occurs, the "failover" process transfers the processing from the failed component to the backup component. The failover process takes over the system re-allocating resources to recover failed transactions and restores the system without loss of data. The "fallback" or recovery stage should follow immediately after, trying to restore the failed component and to bring it back on-line. Ideally this process should be completely automatic and transparent to the users. Databases are the back-ends of most systems today; the need for redundant/replicated and easy to recover databases, without loss of information is one of the problems that exist in providing high availability for systems. Depending on the degree of availability required, duplicating the hardware and/or replicating the database could insure continuous service.

There are different methods to replicate a database ranging from standby database to active replication. For insuring that a database has a consistent replica there are a number of solution available on the market. Problems arise when failed database needs to be restored and brought back on-line. In the simplest failover situation two systems are participating, one is the primary and the other one is the secondary. If the secondary server is sitting idle while waiting to take over, it is considered passive, if it is occupied with server tasks of its own while waiting to take over it is considered active.

The failover schemes are differentiated as follows, by the readiness of the standby system to take over in case the primary system fails:

- *Cold failover*: failover begins when a node fails and the second node is notified to take

over: the database is started and the recovery process begins. This is the slowest failover approach.

- *Warm failover*: When the failover begins the second node is already operational, but some overhead is involved to synchronize the new node with the operations and state of the failed node.
- *Hot failover*: The second node is immediately ready to act as the production node if there is a failure.

When using duplicate systems, we need to insure that the two systems are synchronized at all times. The common forms of replication are file-based replication, database replication and disk block replication. Overall, replication is well studied and various solutions exist, a more delicate subject is fine-grained synchronization in case of high throughput and fallback techniques.

In the next chapter we examine the method used to provide HA for database systems. We will concentrate on database replication, synchronization and recovery techniques.

3. High Availability Methods

The most common approach for achieving high availability is to endow the system with redundant components. The hardware and software components of a system can be made completely redundant but the cost will increase dramatically, thus high availability needs to be achieved within reasonable limits. In the real world there are no perfect solutions, just solutions that are best suited for certain problems. For each of these problems, the industry has provided various solutions, depending on the exact type of cost/performance scenario.

3.1. Hardware HA Solutions

Hardware producers have approached the problem, by making redundant components function together (disks, controllers, power supplies etc.) and then moved to the second step involving hardware/software embedded approach such as Storage Area Networks (SAN) – private networks for storage, or Server Clusters – servers grouped together that appear as single server.

Hardware solutions that aim for "fault free" use technique like disk mirroring, RAID (Redundant Array of inexpensive disks) and Server Clustering. These techniques, which are part of the first generation of high availability solutions, can provide only from 1 to 3 nines in the 5 "nines" method of defining availability[1].

3.1.1. Redundant components

A basic level of availability protection is provided to a system by using components that allow the server to stay operational longer (i.e. uninterruptible power supplies (UPS), Redundant Array of Inexpensive Disks –(RAID), etc.)0.

While these techniques provide protection from hardware failures, they offer little or no protection for the application or networked environment. This is the lowest cost of high-availability protection. However, these kinds of solutions are used for most of the production systems existing today, as a basic protection in case of hardware failure.

RAID technology guarantees disk protection, through different techniques such as disk mirroring, disk strapping, disk spanning, etc. These techniques are largely used to protect special files or entire applications. The level of redundancy (e.g. RAID 0, 1-7, 10 etc.), which is a combination of physical disks, logical disks and controllers, allows a wide variety of choices depending on cost/performance constraints0:

In the database world the most often used techniques are RAID 1 for special files for example control files, and archive or redo log files; RAID 5 or 10 can be used for datafiles.

3.1.2. Clustered Servers

Clustering is a technique for allowing multiple servers to share the workload and in case of failure take over the workload. From the Client's point of view, the cluster appears as a single server; behind the scenes, things are not that simple; however, the clustering technology is mature enough to monitor the different levels involved. At a quick glance,

clustering can be divided into network clustering, data clustering and process clustering0.

- Network clustering deals with managing the network interface to the clusters, which includes IP failover, device monitoring, heartbeat, load balancing etc.
- Data Clustering means that data is available to one or multiple nodes. Either shared storage or non-shared storage can be used; each of these scenarios needs a different solution for the transfer of storage control during the node transition usually provided by the database clustering software.
- Process Clustering deals with allowing multiple nodes to participate in processing of a single application.

Most of the main hardware providers starting with Digital Equipment in the 1980s have embarked onto the quest of clustered systems, promising uninterrupted service and no data loss. Overall, the results represent a big step ahead towards achieving the desired continuous availability.

Figure 1 shows an example of clustered servers where sanity of the system is determined by sending redundant signals between server nodes called "heartbeats".

Providers like SUN propose SUN Cluster 3.0, which is a cluster solution that uses proprietary hardware and Operating System 0. The main key features of this solution are: Global Devices, Global File Service and Global Network Services that enable multiple Solaris systems to function together.

Compaq's OpenVMS Cluster can have components up to 80 KM apart reducing the risk of disaster related failures 0.

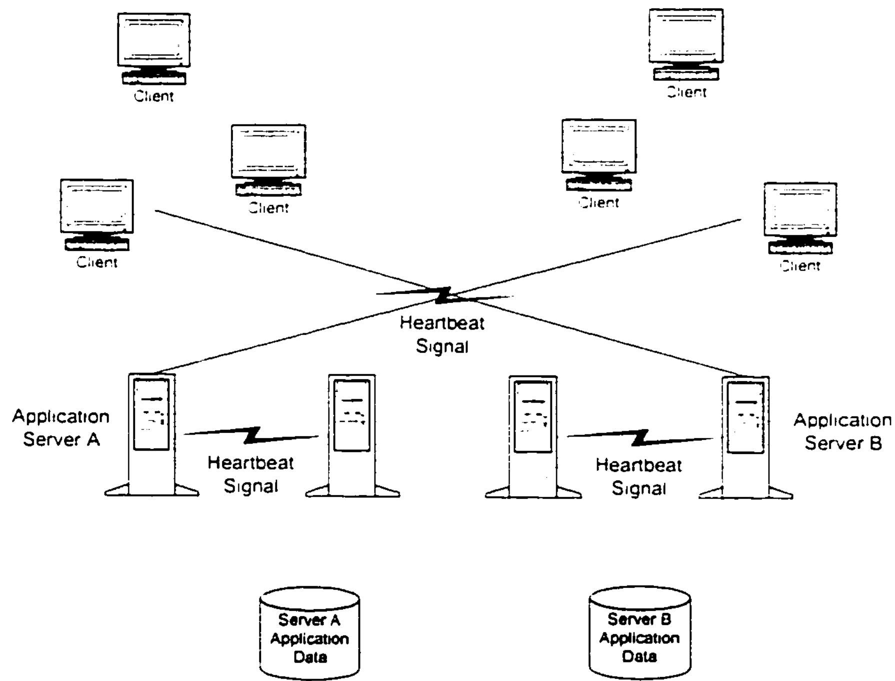

Figure 1 Clustered Servers

HP proposes a middleware software solution in the form of HP Cluster Extension XP. It brings together heterogeneous systems like Veritas Cluster Server on SUN Solaris. HACMP on IBM AIX or Microsoft Cluster on Windows 2000 [6]. Through its disk arrays HP provides fast failover using array-based mirroring host platform-independent solution offloading servers of replication tasks. Hence. this cluster solution can be extended over metropolitan distances. not only campus wide.

Veritas has put together a cluster in a solution called Veritas Cluster Server that works on multiple platforms like SUN Solaris. HP/UX. and Windows NT[8]. The features provided by Veritas are similar to other products such as scalability (up to 32 nodes). flexible failover possibilities (one-to-one. any-to-one. any-to-any. one-to-any). dynamic choice of failover etc. But what is worth mentioning is the integration with other Veritas products such VERITAS Database edition for Oracle. This gets us closer to the problem that

we are facing regarding databases even in a clustered architecture, which is data integrity and data replication for databases. Veritas File System uses Quick I/O files that have become Cache Quick I/O files in the Veritas Database Edition. The Quick I/O files make the database administration more flexible at the OS level, and improve database performance (e.g. faster restart etc). Replication is made a lot faster, because only the changed data blocks at the system level are replicated.

The PC world has started to use cluster solutions, which allows smaller applications to benefit from clustering. One example is LifeKeeper [9] from SteelEye which provides a sophisticated solution using proactive protection, trying to detect faults in advance. It also uses intelligent processes and multiple LAN heartbeats, trying to limit the unnecessary failovers. One of the important features of LifeKeeper is that enables continuous operation during planned downtime for maintenance or upgrades as well as in the event of a system failure or if application ceases to respond.

Overall, cluster parallel processing offers several important advantages. Every machine can be a complete system, used by a complete range of applications. Most of the hardware needed to build a cluster sells in high volume with low prices. In addition, clusters can scale to very large systems, and with little work, many machines can be networked. And most important is that replacing a failed component of a cluster is trivial compared to fixing a part of a failed SMP: thus reducing the downtime [14].

3.1.3. Storage Area Networks

Another approach used in achieving high availability is through hardware protection using Storage Area Networks (SAN) that group together servers and storage devices. This avoids attaching storage to an individual server, which increases the risk of failure. SANs are designed to protect the files throughout an enterprise, using fiber optics connectivity, redundant components and failover technology. This also increases the scalability, reliability and manageability of a system.

As shown in Figure 2 [10] the SANs use new technologies to connect a large number of servers and devices. The deployment of SANs today are exploiting the storage-focused capabilities of fiber channel. The fiber channel SAN consists of hardware components such as storage subsystems, storage devices, and servers that are attached to the SAN via interconnect entities (host-bus adapters, bridges, hubs, switches).

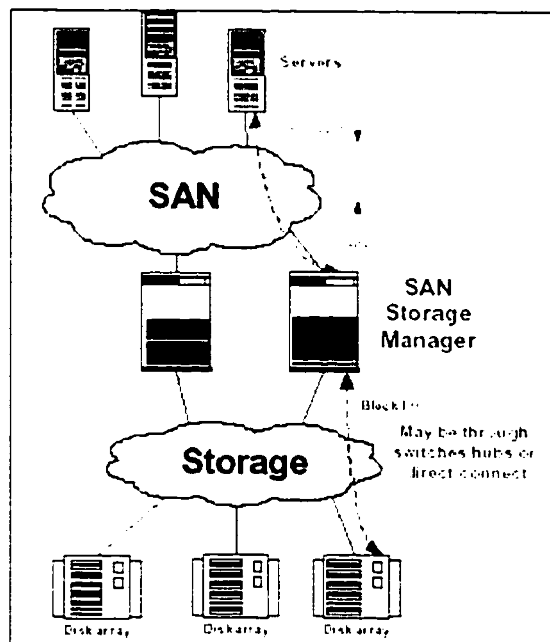

Figure 2 Storage Area Network

The management software is also a major element of storage area networks and can be categorized into two areas: the management of the fiber channel topology or storage network management and the management of the storage. The heart of the storage management software is the virtualization. The storage unit or data presented is decoupled from the actual physical storage where the information may be contained. Once the storage is abstracted, storage management tasks can be performed with a common set of tools from a centralized point, which will greatly reduce the cost of administration. SANs can be proprietary to the hardware vendor like SUN, IBM, Hitachi, or can provide for heterogeneous SANs like HP that can integrate together various platforms and OS.

Although this kind of hardware is much too elaborate and expensive for the problem tackled by this project, it represents one of the hardware innovations providing high available systems; hence worth mentioning.

3.2. Database HA Solutions

For systems using databases, the hardware solutions are not enough to achieve high availability. The most common approach is to create a database replica that can be used as an alternate repository in case of failure. In today's market, there are two major approaches for creating a replica of a database: asynchronous replication or "after event" update of the copy database and synchronous or redundant write to two systems.

The asynchronous replication usually is a built-in database feature and makes use of the transactions logs that are sent to the other backup machines and applied online. Another

method used for asynchronous replication is via triggers/ snapshots, which are able to update all the defined objects to different databases.

The synchronous replication uses the two-phase commit protocol that can be a built-in feature of the database or a middle-tier can be used to ensure that the transactions are committed or rolled back at all sites.

All of these methods in one way or another create multiple copies of the database that can be used in case of failure.

3.2.1. Standby Database and Distributed Databases

In traditional systems, the replication is achieved by having a standby system, which is a duplicate of a production database. The standby replica is updated after the event, thus making the standby system very close to the primary system.

When a failure of the primary system occurs, the standby system takes over and continues the processing. Synchronization of the two databases has to be performed and running transactions have to be rolled back and restarted. At best, the time necessary for this operation is in the order of minutes and in worst case it may take hours before the databases are synchronized.

While the Standby database is running, the primary has to be recovered to reduce vulnerability of the whole system. In some cases the two databases are switched back when the primary has been restored. In other cases the standby database becomes the primary. The Standby approach is intended to offer protection afforded by redundancy, without the constraints of the synchronous updates or the delayed backups. By providing asynchronous,

reliable delivery, applications are not affected by the operation of the standby system or the availability of the standby system.

One of the advantages of such a system is the ability to quickly swap to the standby system in the event of failure, since backup system is already online. Also this system can be configured over the wide area network, which provides protection from site failures. Data corruption is typically not replicated since transactions are logically reproduced rather than I/O blocks mirrored. Originating applications are minimally impacted since replication takes place asynchronously after the originating transaction commits. The standby copy is available for read-only operations, allowing better utilization of the backup systems.

Some of the limitations of this kind of system are that the standby system will be out of date by the transactions committed at the active database that have not been applied to the standby. Also the client application must explicitly reference the standby if the active system fails and they need to be restarted in case of failure. Protection is limited to the database data but the datafiles are not protected. As for the network, adequate network bandwidth is necessary to insure the transfer of logs. Oracle first addressed this problem of asynchronous replication with their Standby Database for Oracle 8i (see Figure 3) and then the improved Data Guard for Oracle 9i. Oracle Standby Database provides a classical solution for log based asynchronous replication that could be managed automatically or manually (copy and transfer of the logs).

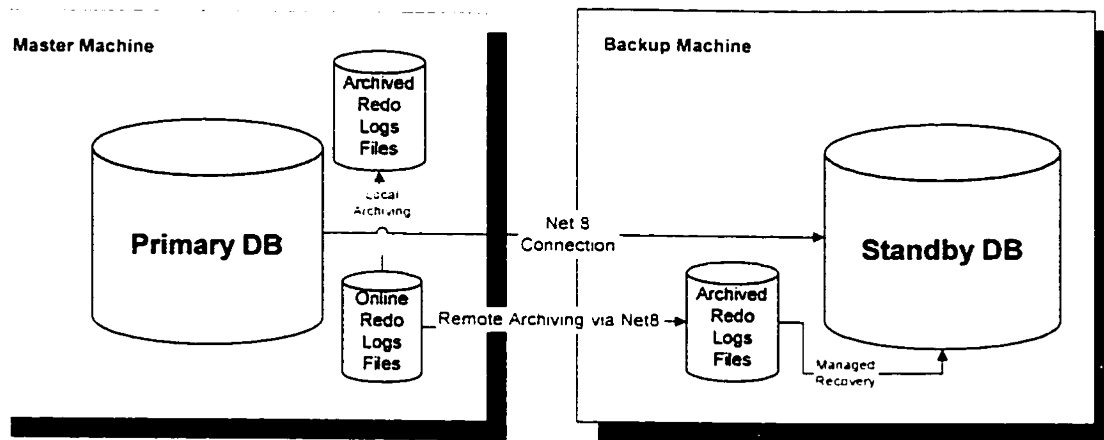

Figure 3 Standby Database Replication

The primary database has to be in archive mode and the archive logs are sent to the standby and applied to this database then the control files are updated.

However this scheme has a disadvantage: at fallback the Standby database cannot become primary and it needs a shutdown of the whole system to revert to the initial situation. More interesting is the Oracle Data Guard that evolves from the initial [36] Standby database. Data Guard allows different methods of replication like synchronous data copy, using two-phase commit of the local database and the standby, or immediate data copy mode (asynchronous mode) and finally batch copy of the redo logs.

Sybase with their Replication Server for Adaptive Enterprise Server provide the Warm Standby that tackles the problem by monitoring the transaction logs and "pushing" the transactions to the standby database. Sybase uses the snapshot technology: instead of sending the logs to the standby and applying them it sends the transactions directly. This technology is widely used especially for maintaining distributed databases where a single transaction updates multiple databases. However, Replication server and Adaptive Server Enterprise Agent Thread will not distribute a multi-database transaction as an atomic unit of work. The single multi-database transaction will be distributed to the replicate Adaptive

Server Enterprise DBMS as a set of transactions, each operating within their respective database at the standby site. While the replication Server guarantees sequential execution of transactions from a single database, it will not guarantee sequential integrity of multi-database transactions (it does not use a dual commit protocol).

DB2 from IBM provides the standby capability as a failover option that can be configured as Idle Standby and Mutual Takeover. In Idle Standby configuration a system is used to run a DB2 instance, and the second system is in standby mode, ready to take over. In Mutual Takeover configuration, each system is designed to backup the other system.

Another form of replication is achieved through distributed databases that encompass multiple database server nodes where each node runs a separate database, each with its own dictionary. Replication is the process of copying and maintaining database objects in multiple databases that make up a distributed database system. One of the advantages of having distributed databases is the ability to replicate across multiple platforms. This kind of replication improves the overall availability of a system by making the data available at multiple sites. If one site becomes unavailable then the data can be retrieved at other sites.

For distributed database scenarios, Oracle provides replication utilities like Oracle Replication Manager, that use two types of replication, master and snapshot replication to replicate database objects [22]. Hybrid architecture between master and snapshots can be used, to meet some special application requirements.

Master replication or peer-to peer allows several sites to manage groups of replicated database objects. For this kind of scenario Oracle uses asynchronous replication and transaction queue. The deferred transactions are pushed at the replicated sites at regular

configurable intervals.

Snapshot replication provides a point-in-time image of a table from the master site to the replica and it can be read-only or updateable. The snapshot needs to be refreshed at certain time intervals to make sure the data is consistent with the master. The changes to the master table are recorded in a table called snapshot log. The main idea behind Oracle's distributed database systems is database link. A database link is a pointer that defines one-way communication between two databases and allows a client connected to one of the databases to access information from both of them as one logical unit. Oracle allows distributed databases to be homogenous, with two or more Oracle databases, or heterogeneous where at least one of the databases is not Oracle (using heterogeneous agents and transparent gateways) and client-server database architecture.

Informix Enterprise Replication is built around the Dynamic Scalable Architecture (DSA), which means that various replication models can be used such as: master/slave, workflow and update-anywhere.

DSA uses a log-based transaction and capture mechanism as a part of the database system. Informix's ER encompasses two different functionality, such as creating a hot standby database (master/slave architecture) and also asynchronous replication of data to one or multiple secondary sites (peer to peer architecture).

In a Master/Slave ownership, there are again three scenarios[34]:

- Data dissemination where data is updated in a central location and then replicated to regional read-only sites.
- Data consolidation where data sets can be updated regionally and then brought together in a read-only repository on the central database server

- Workload partitioning gives the flexibility to assign ownership of data at the table partition level.

Peer to peer update, unlike master/slave ownership where replicated data is read-only, creates a peer-to-peer environment where multiple sites have equal ability to update data. To resolve update conflicts, this replication architecture supports a wide range of conflict detection and resolution routines.

From the above, we can conclude that regardless of provider there are just a few replication methods that are largely used [15]. The so-called first generation technology is variously called "change-capture" or "store and forward" and "log based" methods. These techniques require that a replication tool stored locally at each site captures the changes to data : these changes are forwarded to other sites at replication time. The second generation of replication technologies involves the use of "direct to the database" methods, which examine only the net data changes that have taken place since the last replication cycle.

3.2.2. Parallel Processing, Clusters

Parallel Servers are the database built in capability to synchronously replicate the transactions processed by a database system. A database instance is running on each node and the data is stored on separate storage. The workload is distributed among the different nodes belonging to the Parallel Sever or Application Cluster.

This database solution comes on top of the hardware clustering previously discussed and deals with the application issues. It therefore allows multiple instances to work together, share the workload and access the storage. The clusters share disk access and resources that

manage data, but the distinct hardware cluster nodes do not share memory[21]. Clustered databases could be either shared disk or share nothing databases:

- Shared disk approach is based on the assumption that every processing node has equal access to all disks (see Figure 4). In pure shared disk database architecture, database files are logically shared among the nodes of a loosely coupled system with each instance having access to all data.

The shared disk access is accomplished either through direct hardware connectivity or by using an operating system abstraction layer that provides a single view of all the devices on all nodes. In this kind of approach, transactions running on any instance can directly read or modify any part of the database. Such systems require the use internode communication to synchronize update activities performed from multiple nodes. Shared disk offers excellent resource utilization because there is no concept of data ownership and every process node can participate in accessing all data. A good example of shared-disk architecture is Oracle Parallel Server (OPS) that constitutes the classical approach for this kind of architecture [18]. OPS offers protection against cluster component failures and software failures. However, since OPS as a single instance Oracle, operates on one set of files, media failures and human error may still cause system "downtime". The failover mechanism for OPS requires that the system has accurate instance monitoring or heartbeat mechanism. The process of synchronizing requires the graceful shutdown of the failing system as well as an accurate assumption of control of resources that were mastered on that system.

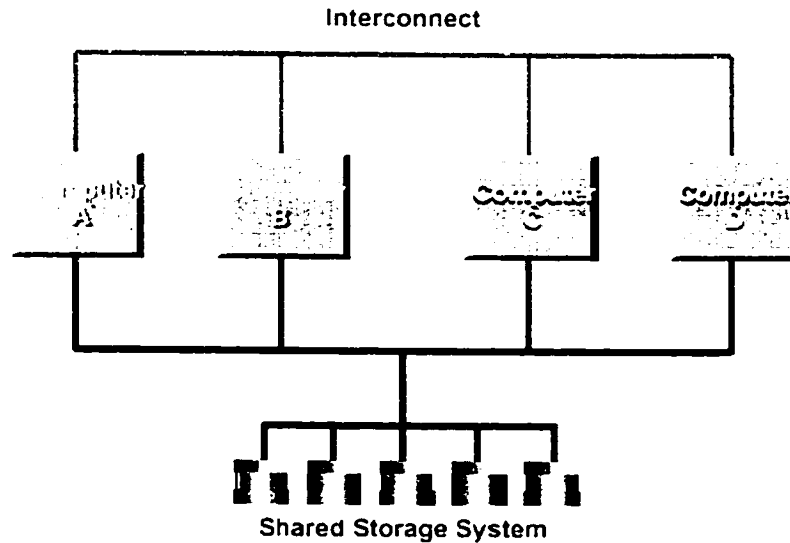

Figure 4 Clustered Database- Shared Disk Architecture [18]

As for the clients connections, the Transparent Application Failover enables an application user to automatically reconnect to a database if the connection breaks. Active transactions roll back, but the new database connection, made to a different node is identical to the original one. Hence, we can say that the client sees no loss of connection as long as there is one instance left serving the application.

- In pure shared nothing architectures shown in Figure 5, database files are partitioned among the instances running in the nodes of a multi-computer system. Each instance or node has affinity with a distinct subset of the data and all access to this data is performed exclusively by the dedicated instance. In other words, a shared-nothing system uses a partitioned or restricted access scheme to divide the work among multiple processing nodes. Parallel execution in a shared nothing system is directly based on the data-partitioning scheme. When data is accurately partitioned, the system scales in near linear fashion [34]. Multiple partitions are accessed concurrently, each by a single process thread.

A transaction executed on a given node must send messages to other nodes that own the data being accessed. It must also coordinate the work done on the other nodes that perform the required read/write activities. However, shared nothing databases are fundamentally different from distributed databases in that they operate one physical database using one data dictionary.

Informix Parallel Extended Dynamic Server proposes a shared-nothing architecture through partitioning of data, partitioning of control and partitioning of execution.

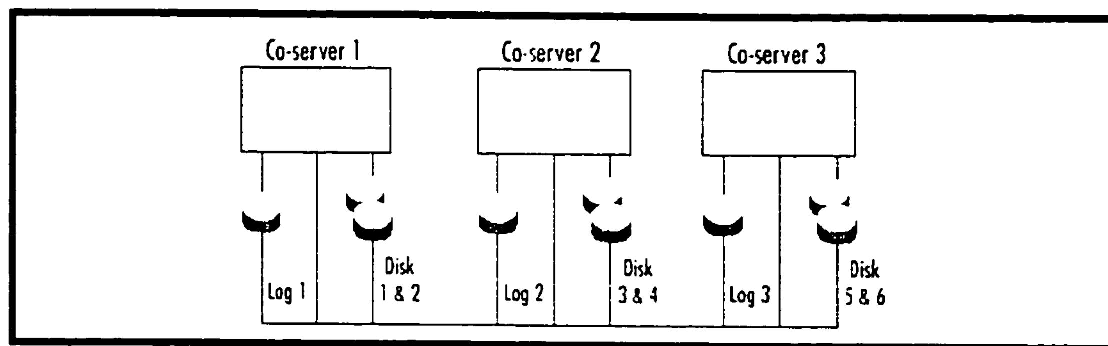

Figure 5 Clustered Database-Shared Nothing Architecture [34]

In their case, each node of the cluster runs its own instance of Informix Dynamic Sever that consists of basic database services for managing its own logging, recovery, locking and buffer management. This instance is called a co-server. Each co-server owns a set of disks and the partitions of the database that reside on these disks. A co-server will typically have physical accessibility to other disks owned by other co-servers to guard against unexpected failures, but in normal operation each co-server will access only those disks that it owns. In case of failure of a node, there is no easy way to recover without shutting down the database, hence this solution provides means for parallel execution and load balancing but does not truly guard against failure.

A very interesting solution of shared nothing architecture and very high availability

is provided by Clustra database. it is not that well known except in the telecommunication world. Clustra is a traditional database server. in the sense that it manages a buffer of data with a disk-based layout in blocks: it has a B-Tree access method. a semantically rich two-phase record locking protocol. and it has a two-level logging approach. However. it is main memory-based in the sense that tables may be declared to reside in main memory. It ensures high availability [20] by dividing data into fragments that are again stored in data processing and storage units. In turn. the nodes are housed in what it is referred to as data redundancy units. If a user needs to add nodes. the system scales linearly. It also automatically repairs the data of corrupted or lost nodes. and provides optional online. spare nodes for maximum availability.

This database addresses also the planned outages through rolling upgrades and online schema modification (OSM). Total transaction capacity increases linearly with number of nodes in the system. When a greater capacity is needed. new nodes can be added: the capacity for each node stays the same. while the total capacity increases. Scaling of the database is also linear. If the number of nodes is doubled. the storage capacity is double. if the nodes run on identical hardware.

The Clustra database runs over cluster off-the-shelf hardware. but doesn't require special clustering features or operating system software that traditionally add complexity to the system management and integration. The distinct feature of this architecture is that the nodes do share neither disks nor memory. This keeps nodes isolated from one another: so failed nodes can be replaced without involving others. A node is defined. as a computer comprised of a CPU. local disk and main memory. Nodes are linked together to form a fully replicated logical database via a LAN. The database is fully capable of repairing itself when

a failed node will not restart or the database content has been corrupted, without any disruption in system operation. This capacity of self-healing is perhaps what is most remarkable about this database.

- The shared-cache architecture provides the benefits of both shared disk and shared nothing databases without the drawbacks of either architecture. This solution is based on a single virtual high performance cluster server that utilizes the collective database caches of all the nodes in the system to satisfy application request to any one node. In this way it reduces the disk operation necessary for inter-node synchronization. Traditionally shared disk database systems use disk I/O for synchronizing data access across multiple nodes. The cache fusion architecture overcomes this weakness by utilizing Global Cache Services for the status and transfer of the data blocks across the buffer caches of the instances. Real Application Clusters is the next generation of Oracle Parallel Server and continues the pursuit of insuring almost continuous availability by hiding failures from the users and application server clients.

The aim of the clustered systems in general is to offer transparent application failover by redirecting the clients that are connected to a failed node to available nodes. This is done either directly by the cluster software through configuration or by simple application coding techniques through the client failover libraries.

Fault resilience is achieved in clustered databases through re-mastering all database resources onto the surviving nodes, guaranteeing uninterrupted operation as long as there is at least one surviving database node.

3.3. Transactional HA Solutions

In the pursuit of having data replicated at different sites the Transaction Processing approach is most commonly used: it is a way to coordinate business transactions that modify databases and keeps a write-ahead-log of all the modifications made to the database over a period of time. It is advisable for databases that are constantly modified, to ensure that the data modifications are properly stored. If an error occurs or the system crashes while modifications are being made, the write-ahead-log can be used to restore the database to a previous error-free state.

The purpose of the present project is to find an economical solution for a HA system, which provides fast fallback and restart in a fast changing transactional system using off-the-shelf components.

3.3.1. Transaction Processing

A transaction is used to define a logical unit of work that either wholly succeeds or has no effect whatsoever on the database state. It allows work being performed in many different processes at possibly different sites to be treated as a single unit of work. Transaction data can be stored in a flat file or be controlled by a Relational Database Management System where strict rules are applied. Data within a Relational Management System must adhere to the ACID properties [25] to avoid undefined behavior:

Atomicity: A transaction's changes to a state are atomic, either all or none of the changes made in the data happen. This means that all parts of the transaction must be

complete. If an incomplete transaction is interrupted or cannot complete, the entire transaction is aborted

Consistency: A transaction is a correct transformation of a state. This means that data must be consistent within database before at the end of each transaction.

Isolated: Even though transactions execute concurrently, it appears to each transaction that others are executed either before or after it. Another way of saying this is that transactions are serialiazable.

Durable: Once a transaction completes successfully, its changes to the state survive failures. Even if the database fails the changes should be reflected in the system after it is restored.

In the distributed transactions processing, shared resources such as databases are located at different physical sites on the network. A transaction-processing monitor helps to facilitate distributed transactions processing by supplying functions that are not included in the OS. These functions include: naming services, security at the transaction level, recovery coordination and services, fault tolerance features, such as failover redirection, transaction mirroring and load balancing.

Because work can be performed within the bound of a transaction, on many different platforms and involve many different databases from various vendors, a standard has been developed to allow a manager process to coordinate and control the behavior of databases [24]. X/Open is a standard body that developed the Distributed Transaction Processing Model and XA interface to solve the heterogeneous problem.

X/Open applications run in a distributed transaction-processing environment. In an abstract model, the X/Open application calls on Resource Managers (RMs) to provide a

variety of services. For example, a database resource manager provides access to data in a database. Resource managers interact with a Transaction Manager (TM), which controls all the transactions for the application. The *X/Open DTP Model* defines the communication between an application, a transaction manager, and one or more resource managers. The most common RM is a database (i.e. Oracle, DB2, Sybase etc.). The *X/Open XA* interface is a specification that describes the protocol for the transaction coordination, commitment, and recovery between a TM and one or more RMs.

3.3.2. Transactional Systems

A number of transactional systems on the market are used as a middleware in a three-tier architecture for distributed transaction-processing systems. As an example, we can look at Customer Information Control System (CICS) and ENCINA from IBM, and TUXEDO developed by AT&T Bell Laboratory.

CICS is considered as IBM's general-purpose online transaction processing (OLTP) software. It represents the parent of all transaction processors[29]. CICS is a layer that shields applications from the need to take account of exactly what resources are being used, while providing a rich set of resources and management services for those applications. In particular, CICS provides an easy-to-use application programming interface (API), which allows a rich set of services to be used in the application and to be ported to and from a wide variety of hardware and software platforms where CICS is available. CICS is a very general, all-purpose transactional system, used for communication with devices (terminals), including printers, workstations and also interconnects with other CICS or non-CICS

systems.

Another transactional system is ENCINA that specializes in. providing means for building distributed transactional applications. The foundation of the ENCINA environment is the ENCINA Toolkit. which is a set of low-level components for building distributed transactional applications. ENCINA provides higher-level interfaces that are built on top of the Toolkit. These interfaces hide many of the complexities of Toolkit-level programming. The higher-level interfaces used for writing transactional applications include Transactional-C and the ENCINA TX interface. ENCINA also supplies a transactional interface (CPI-C/RR) for writing X/Open-compliant applications that use the Peer-To-Peer Communications Services (PPC). ENCINA provides the APIs necessary to communicate with different RMs such as databases. but it does not particularly provide a direct interface with the most used database such as: Oracle. Sybase. Informix etc.

TUXEDO on the other hand. is very versatile allowing the users to build and manage 3-tier client/server applications for distributed mission-critical applications[28]. It supports server components executing in the network environment. Component software applications are distributed and characterized by a logical 3-tier architecture:

- Client application form the first logical tier. initiating and invoking services for core business processing functions such as database reads and updates.
- The middle tier is composed of managed server components: server components advertise their named services. process incoming message-based requests for these services. and return the results to the requestor—a client or another service
- Resource managers. such as relational databases. constitute the third tier. and manage the application's data assets.

Tuxedo provides the underlying execution environment for 3-tier applications. managing client access to server components and constituent services. managing the service components themselves. and providing the point of integration for database resource managers.

Through standard interfaces. Tuxedo is easily integrated with the leading databases (i.e. Sybase. Oracle). file and queue resource managers. There are two key areas of integration:

- Usage of APIs to perform standard manipulation functions (i.e. embedded SQL pre-compilers);
- TM integrates with the resource managers X/Open DTP XA[25] interface for global transaction coordination. The XA interface is transparent and encourages database independence by enforcing a clean separation between business logic and the data model.

For distributed transaction processing with the goal of achieving database replication. TUXEDO represents one of the best candidates for a middle-tier. due to the programming ease and well-defined architecture.

Overall the transactional systems are a viable solution largely used in the distributed systems today such as banking applications. airline reservation systems etc. They allow building real-time. scalable and complex application.

3.4. HA Methods - Conclusions

The HA solutions presented above represent the state of the art in the industry and are

largely used in various applications. They each represent a solution for a particular system need. In our case most of the solutions are ruled out by their complexity, elevated cost or lack of features to provide the HA that we are aiming for being less than 5 minutes of downtime per year.

If we look at the hardware solutions presented, the Clustered Servers and Storage Area Networks are far too complex and expensive for small or medium size applications that we are aiming for this prototype. However, in any production system there is a need to use redundant components (e.g. CPU, disks, controllers etc.), to avoid a single point of failure.

As for database replication, the only solution that meets our high availability criteria, is the clustered databases, having the nodes synchronized and able to take over in case of failure. This solution it is again very expensive and proprietary to one vendor, not allowing heterogeneous databases to be clustered.

This brings us to the idea of finding a solution to maintain to identical copies of a database, which is not proprietary to one vendor and where the transactions are coordinated from outside of the database. As presented above, the transactional systems are largely used for building distributed applications; hence using them to synchronize databases is just an extension of their capabilities.

4. Replication Solution for ensuring HA in an OLTP environment

4.1. Problem Definition

Choosing a data replication technology can be a difficult task, due to the large number of products on the market with different implementation and features. To add to this complexity, data replication solutions are specific to a DBMS, file system or OS. Making replication decisions depends first of all on the amount of data that can be lost.

- If minutes of lost transactions are acceptable, an asynchronous solution will probably provide a more cost-effective solution while still offering fast recovery. The most common method is shadowing where changes are captured from the primary site and applied to the recovery site.
- If this is unacceptable then synchronous replication can be used to mirror the two databases, where the changes are applied at the secondary site in lock step with changes at the primary site. In this scenario, only the uncommitted work is lost.
- When, no work can be lost the next step is to use transaction-aware replication. The primary advantage of this approach is that the replication method understands units of work (e.g. transactions) and the data integrity has a greater potential.

Problems arise in a transactional environment such as the telecommunication world, where no data can be lost and even the uncommitted/rolled back transactions have to be reapplied. The solutions that exist and utilize the synchronous replication and two-phase commit are database built-in features (e.g. Parallel Servers) that are proprietary to one vendor and usually function in a homogenous environment.

Considering a system with two databases that are updated at the same time using synchronous replication. the purpose is that. in case of a failure. to be able to resynchronize the two databases. without shutting down the system.

The problem addressed by this project is finding a solution. for a fast fallback and restart in a fast changing transactional environment.

The part of the problem that is not addressed by the traditional Standby systems is the fine-grained synchronization after the failed system is recovered. In any database system. we can duplicate a database up to a point in time. but the synchronization can't be done completely while the system is running without causing data inconsistency.

By using transactional systems in conjunction with the database. the problem of synchronous writes in two or more databases is handled by the XA interface[25] The recovery phase in such case. where a failed database has to be resynchronized and restarted is tackled by this project.

4.1.1. Assumptions and Constraints

For this project. we will be looking at three-tier architecture. consisting of an Application Server that connects to a database through a middle-ware and various Client Applications. who access the Application Server.

The following are some of the assumptions and constraints:

- The Application Server handles the logic and all the changes done to the two databases.
- The database is used strictly as a repository: hence the logic is handled outside the database.

- The Client Applications connect only to the Application Server that handles the entire interaction with the database.
- The synchronous replication is handled by the middleware (e.g. TUXEDO).
- Changes related to the database schema, or software upgrades are not addressed in this project.

4.1.2. Detailed Problem Definition

In case of failure we will be looking at the following two scenarios in order to restore the database:

- Failsafe

If one of the databases fails, the Application Server will detect the malfunction and switch the system to a degraded mode. This means that all incoming transactions are written to the working database and also are logged into a Journal file.

- Fallback

Failed database is restored by making a copy of the functional database, up to the time of the failure (point in time recovery). Then all the changes logged in the Journal are applied to the recovered database. All the new transactions should be kept in queue so no writes occur. The system should be then switched back to the normal mode and activity resumed.

The major issues that need to be addressed are:

- What kind of operations should be stored in the Journal.(e.g. only the ones that modify the data, INSERT , UPDATE etc)
- Gracefully switch between normal mode to degraded mode without loss of data.

- What operations are allowed during the degraded mode functionality (e.g. No schema changes are allowed etc.)
- Applying the missing transactions to failed database.
- Switching back to normal mode from degraded mode.

These issues represent the core of the problem and will be addressed in detail in the design phase.

4.2. Problem Approach and Analysis

Having defined the problem above, this section will present the necessary requirements for the design and implementation of the system.

4.2.1. System and Functional Requirements

The following are the functional requirements:

- Provide the Clients transparent access to the system/databases, while presenting a single, stable interface.
- In case of failure, the Clients connections to the system and ability to work should not be affected.
- Switch gracefully the Clients from one system to another in case of failure.
- Application Server's availability should be decoupled from the database availability.
- The Application Server including the middle-ware and Journal files have to be protected against failure.

- Database access, rather than being made directly through the Application Server will be handled by the transactional system.
- Once a transaction has been taken in charge, and confirmed by the system it must not be lost, even if it is to be processed later.
- Provide queuing mechanism store the transactions in case of a failure and thus to avoid their loss.
- Provide timely response to the Clients and advise if a transaction needs to be rejected.
- When a database failure occurs, provide a graceful failover and fallback procedure.
- In case of database failure, the functionality and capability of the overall system should not be diminished.
- The data stored by the two databases should be identical at any time, except in case of degraded mode.
- All the changes to the database that occur during the degraded mode need to be recorded into a Journal file.
- Data inconsistency should not occur at any time. Hence, the switch to and from degraded mode should not generate any data inconsistency.
- Provide means for switching completely to a backup system, to ease hardware and software upgrades.
- The solution provided should be easy manageable from an administrator point of view.
- The system must allow the use of heterogeneous databases.

These requirements are oriented towards the functionality of the system in case of failover and fallback procedures and the database behavior in these particular situations.

4.3. Architectural and Functional Constraints

This kind of architecture is close-fit for transaction-oriented systems, as a non-expensive solution. However, it has a level of complexity in the middle-tier level and the programming involving XA interface.

The solutions on the market that have replication and failover capability built in the database, are either too expensive or do not provide enough availability (after event replica). Relying only on hardware protection with software monitoring means again that very expensive choices need to be made. Hence, the solution that was chosen where the logic is taken outside of the database is a hybrid between existing technologies and it is not viable in all circumstances.

The major constraints that exist for this kind of architecture are:

- The choice of architecture needs to be based on off-the-shelf products to maintain a reasonable cost for such a system. This refers to both hardware and software products that will be used. As identified before HA can be achieved, using specialized hardware or software that are fault tolerant, but the cost and complexity would be too high.
- The hardware and software monitoring should be insured by third party software that will monitor the system, to insure that no failure occurs.
- The system needs at least two nodes to insure the failover and fallback procedures.
- There is a maximum of databases that can be updated at the same time. The constraint for maximum number of databases is given by the transactional system used, and by the network bandwidth available.
- The number of databases used can impact the performance of the system due to the

slowdown introduced by the synchronous update. This system will function in a LAN environment only.

- This kind of solution is viable only in a three-tier architecture where the Client Application does not have direct access to the database.
- There is an extra burden of securing the Journal, assessing its size and ensuring for space.
- The system is conceived for a transactional environment and it is dependent on the Transactional System.

5. Proposed Solution and Its Design

5.1. Design Rationale

5.1.1. Choice of architecture

The main criteria for selecting architecture are:

- Provide a relatively simple solution for a highly available system that makes use of databases.
- Produce a system that is capable to function in a demanding transaction oriented environment.
- The need to take the failover and fallback logic outside of the database
- Update of two or more databases without losing any data.

Figure 6 shows the proposed architecture, which is a classic architecture for such systems. It is a three-tier architecture including Client, Server and Database as repository.

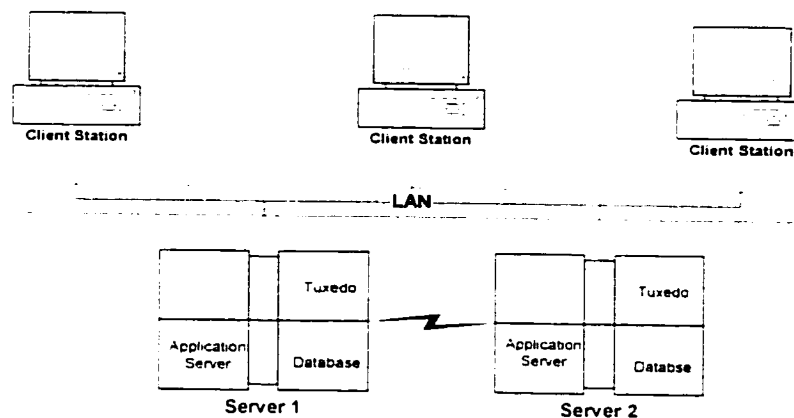

Figure 6 Overall Architecture

The Application Server is build around the middleware and resides physically on the same machine as the database. Each Database and Application Server resides on the same machine, but the two databases have to reside on different machines. The two machines are a mirror of each-other forming a cluster, which is the simplest form of high-availability systems.

5.1.2. Availability

The core of the problem is the availability of the system: hence, the architecture is chosen keeping in mind the "no single point of failure" concept. Therefore, each of the components of the system needs to be guarded against failure:

- The minimum hardware necessary to be able to replicate the servers and the databases is two machines.
- In case of database failure, the failover and fallback processes should be in place.
- The Journal file, the database data files and control files need to be stored on the redundant hardware (e.g. mirrored disks).

We could say that main design decisions were taken based on the availability constraints.

5.1.3. Overall System Cost

This kind of system combines existing affordable technology in order to provide a solution that achieves high availability. Hardware wise, this solution makes use of a simple system composed of two PC like machines without using special hardware that could increase the cost dramatically. As a heterogeneous system where the databases can be different, we can

use one database, which is more expensive (e.g. Oracle) and the second one can be less expensive (e.g. Informix).

The Transactional System introduces an extra cost, which is we can say "the price to pay" for making use of the XA interface and being able to take the replication mechanism outside of the database.

5.1.4. Scalability

The system presented in this paper deals with the simplest case of two machines/databases, just to demonstrate the viability of the solution. The scalability and the flexibility of the system are determined by the Transactional System that can be configured to deal with much more than two databases. Moreover, once the system is designed to deal with distributed transactions, it is just a matter of configuration at the Transactional System to point to additional databases. As for the recovery part the mechanism stays the same in case of databases failure consisting of redirecting the recovery process towards the failed database.

By taking the logic outside the database and decoupling the architecture in three layers, the issue of scalability becomes easily manageable.

5.1.5. Manageability

The aim of this system is to deliver an easy to maintain almost self-recoverable system. The three-tier architecture allows separating the maintenance of the system into the three layers: Client, Application Server/Middle-ware and Database. Once the system is setup, the manual

intervention is limited to the fallback procedure, in case of database or node failure. This is due to the fact that the failed database needs to be restored up to a point in time by the administrator. Even this procedure can be automated to reduce manual intervention.

As for the Application Server, there is no need for manual intervention, since the middleware will migrate at this level all the process from one system to another in case of failure.

5.2. System Architecture

The proposed architecture for the system takes into account all the criteria concerning the architecture quality, discussed previously. For the proof of concept that we are trying to conduct, we will consider minimum of necessary hardware as follows:

- Two PC based Servers with Linux OS. The machines used for testing have Pentium III 700 MHz processors, and 512 MB of RAM.
- TCP/IP Network communication.
- Tuxedo software, which will be used as a Transactional System to build the Application Server.
- Oracle 8I Database as database repository.
- Resource Manager Provided by Oracle necessary to work with Tuxedo's Transaction Managers.
- The Journal Files provided by Tuxedo's Q queuing mechanism, stored on mirrored disk to minimize the loss of data in case of hardware failure.

Note 5-1: The second database can be Informix, to prove that the solution supports heterogeneous databases. However, for this project we will use only Oracle databases.

Figure 13 gives an overview of the system, identifying all the resources and interfaces used. The Servers via Tuxedo's Transaction Manager communicate with the Resource Managers of the two databases or Queues, in order to commit all the changes (e.g. SQL commands) sent by the Client Applications.

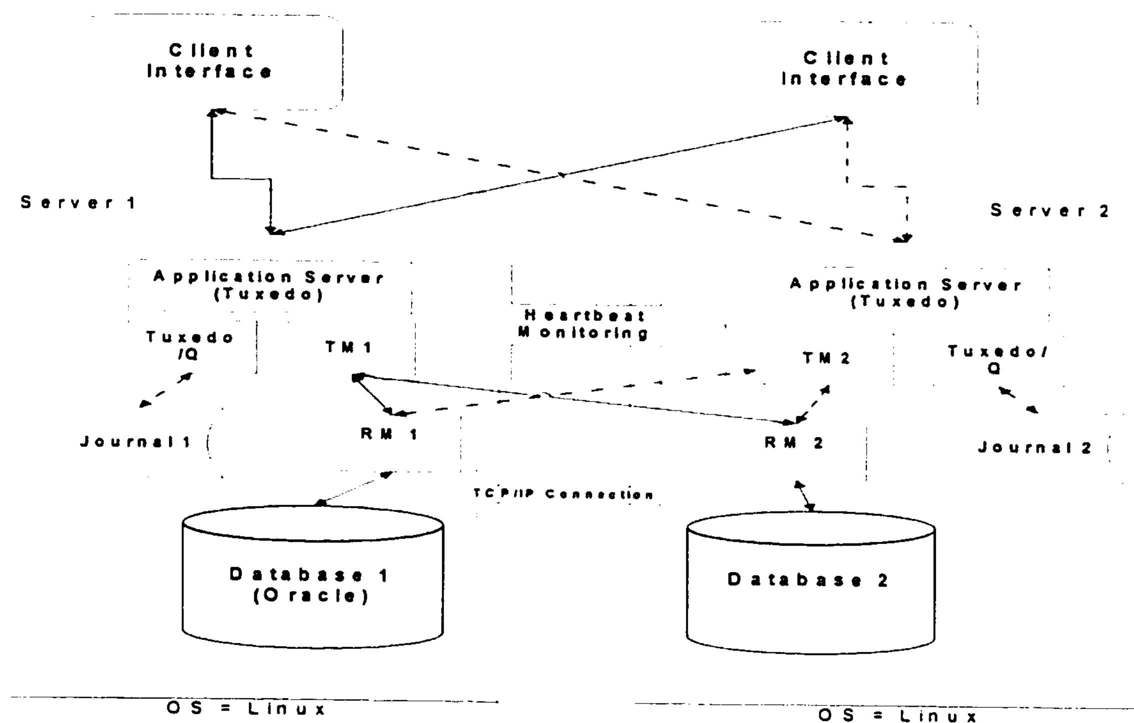

Figure 7 System Architecture

- In case of Database failure, the operation continues with one Resource Manager, and the Transaction Manager instead of the failed database updates the Journal File via the Tuxedo/Q queue.
- In case of Server failure, the application functions in Degraded Mode on one Server.

until the other one is restored. The functionality is similar with the Database failure scenario.

In case of large amount of data the databases can reside on different machines with separate storage, like separate disk arrays etc.

5.3. System Design

The main part of design for this system revolves around the Application Server that makes extensive use of Tuxedo's capabilities to handle transactions in a distributed environment.

5.3.1. Dealing with Transactions

The Tuxedo System provides a C-based interface called Application to Transaction Monitor Interface (ATMI). As shown in Figure 8, the transaction boundaries are defined by the Transaction Manager. The TM provides the Application with API calls to inform it of start, end and disposition of transactions.

The Tuxedo System provides the components for creating the Transaction Manager. The Resource Manager vendor (Oracle, Informix, etc.) provides an XA compliant library that is used along with the Tuxedo utilities to build a Transaction Manager program.

The X/OPEN application does not establish and maintain connections to the database. The database connections are handled by the TP Monitor and XA interface provided by the RM vendor. The RM also provides the means to communicate between itself and the Application via SQL. The TM and RM communicate via XA Interface, which

generally describes the protocol for transaction coordination, commitment and recovery.

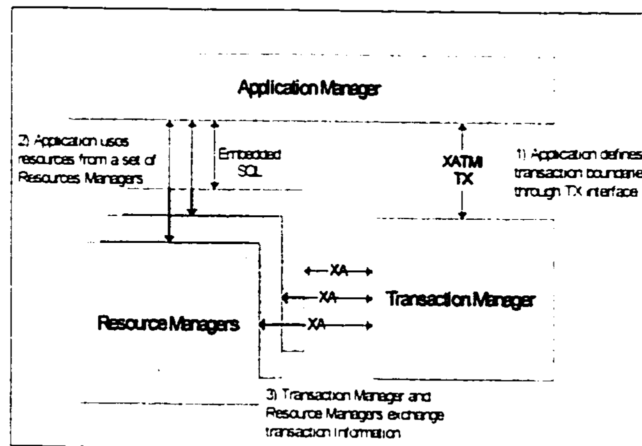

Figure 8 X-OPEN DTP Model

In our case when the system functions normally, a transaction would entail updating two databases via two RDBMS Resource Managers using two phase commit protocol (2PC). In case of database failure a transaction would entail updating one database via RDBMS RM and the Journal via Tuxedo/Q RM.

5.3.2. Subsystem Design

5.3.2.1.Client Design

The function of the Client is to gather information for subsequent processing and display the information. In case of Tuxedo Client the core business functions are accessed as services available to the Client. In our case the Client application will send information to the services to update the databases (see Figure 9).

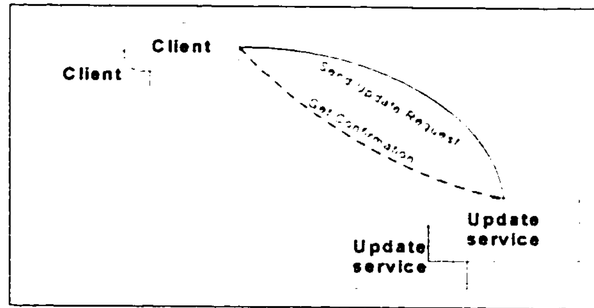

Figure 9 Client Design

The service advertised by the Server will be Update and its functionality will be transparent to the Client in case of database failure, meaning that the Client will always call this Service regardless what happens behind the scenes.

5.3.2.2.Server Design

A Tuxedo Server encapsulates the business logic in the form of Services. The services can be advertised, or made available to the clients, and de-advertised via programming not only the system is booted.. For this application we will use this capability to change the functionality of our main service called Update.

Server applications compiled along with the BEA Tuxedo binaries to produce the server executables. In our case the servers manage the resources, hence update the databases or the queue and also manage the recovery process. The servers are booted along with the Tuxedo system and continue running until they receive a shutdown message. A typical server may perform thousands of service calls before being shutdown and rebooted.

The design of the servers and their service is based on the concept of group that

manages the resources. Figure 10 shows a Server Group that handles the interaction with a database via the ATMI interface. The Server Groups are created by updating Tuxedo's configuration files. and accommodate the transactions related to the database or any other resource manager.

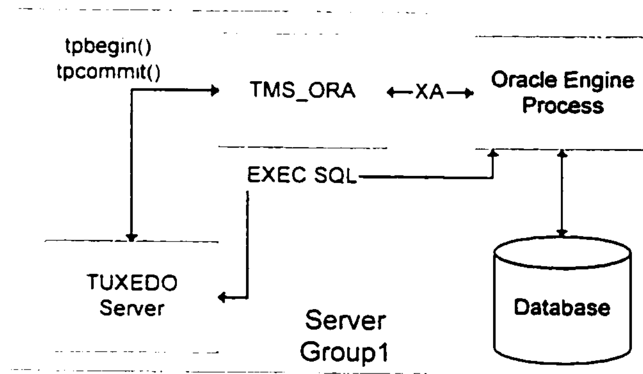

Figure 10 Tuxedo Application Server –Server Group

For each RM that is used. four in our case we need a Server Group to coordinate all the transactions directed towards this resource. The resources are databases and flat files. the latter are be updated via the queuing mechanism provided by TUXEDO. called Tuxedo/Q.

Figure 11. shows an example of five server groups that handle the update of the two databases. the update of the database and a queue and the update of the failed database from the queue. The system needs two Server Groups for accessing the two databases and another two server groups to access the Journals. The fifth one is needed to handle the global transactions generated by the Client. A group is assigned to a specific machine and the servers could run on both machines to handle the failover and also the load balancing.

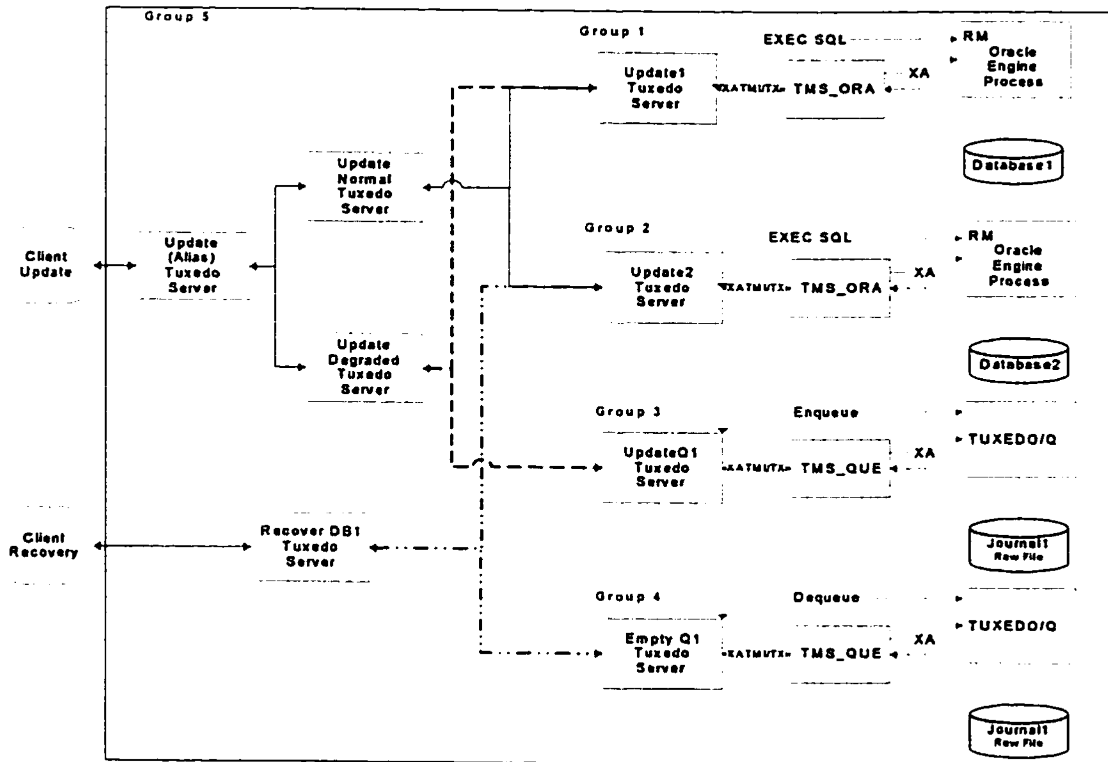

Figure 11 Tuxedo Multiple Server Groups

Having to update multiple resources defining the same transaction, we will use global transactions. In Distributed Transaction Processing environment many RMs have to operate the same unit of work. This unit of work is called global transaction and is logged in the transaction log (TLOG) only when it is in the process of being committed. The TLOG records the reply from the global transaction participants at the end of the first phase of a 2-phase-commit protocol. A TLOG record indicates that a global transaction should be committed; no TLOG record is written for those transactions that are to be rolled back. In the first phase, or pre-commit, each Resource Manager must commit to performing the transaction request. Once all parties commit, transaction management commits and completes the transaction. If either task fails because of an application or system failure,

then both tasks fail and the work performed is undone or "rolled back" to its initial state. The TMS that coordinates global transactions uses the TLOG file. Each machine has its own TLOG.

For our application the Server and implicitly the Tuxedo middleware will be distributed over the two machines as can be seen in the Figure 12.

The Client application could reside on the same machine as the Server application or on workstations. For this project, since we are concentrating the design of the Server, we will choose the more convenient design for the Client: hence it will be a local Client.

The Client applications and the Server applications are distributed over the two machines. From a Tuxedo point of view the configuration depicted in Figure 12 is one of a Master Slave. The Master contains the Distinguished Bulletin Board (***DBBL***), which handles all the global changes, monitors the system and restarts the failed processes. Each machine has its own Bulletin Board (***BBL***) that monitors locally the machine and reports to the DBBL the state of the servers and services.

The second machine is configured as a Backup of the Master: hence the application can be completely switched over in case of the Master machine failure

All the changes in configuration, once the software is compiled, are stored into a binary file, called ***tuxconfig*** that has to be copied on both machines. Tuxedo provides a network listener software called ***tlisten*** that launches the ***Bridge*** server, allowing the two machines to communicate with each other.

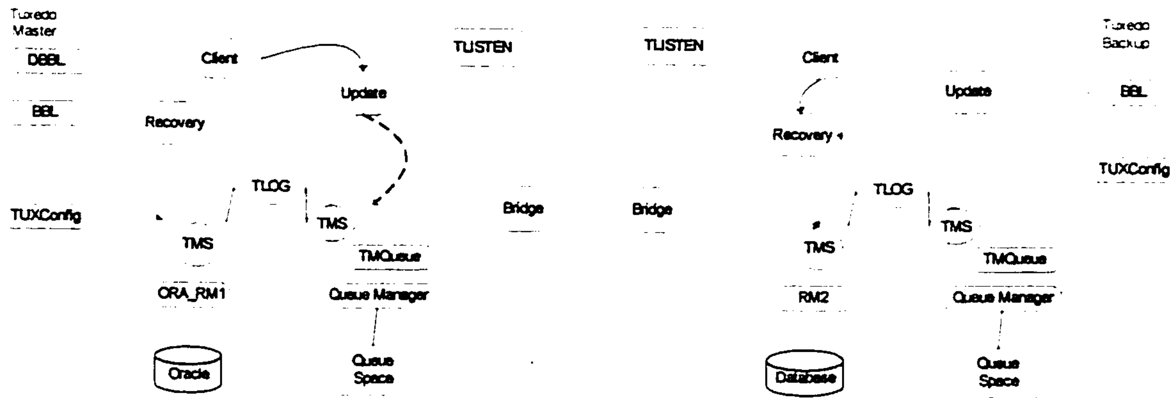

Figure 12 Application Design

The System can find itself in three different states, depending on the database node availability; these states will be mapped into two Services: Update Service and Recovery Service. From a functionality point of view these service are as follows:

- *Update Normal*: Update the two databases via the Resource Managers. The Client application calls the Update service advertised by the server, which result in the update of the two databases.
- *Update Degraded Mode*: In case of database failure the server will advertise a new Update service that will update one database and the queue, which is our Journal.
- *Recovery Mode*: When the database has been restored up to the time of the failure, a new service is called by a special client that updates the failed database from the queue. When the queue is empty it will advertise the Update Normal Service instead of the Update Degraded.

Therefore there are two types of processes: the automatic failover handled by the Update service and the Recovery that will be called by the administrator. The Recovery is started when the failed database in our case the Database 2 is recovered up to the time of

failure. The time of failure is logged by the Update degraded service in a queue and can be retrieved by the administrator via a special Client. When all the messages are extracted from the queue and the two databases are synchronized, the Update degraded service is unadvertised and the Update normal service is advertised again. To avoid adding more messages into the queue while the two services are switched, volatile queues can be used between the Client application and the Update service, making use of Tuxedo's Queue Forwarding mechanism. Hence, both Update1 and Update2 Services are made unavailable, while the Update Normal is advertised; in this way, the incoming messages are stored in the volatile queue before being sent to the Update service, avoiding data inconsistency.

All the services described above are running on both machines, and regardless of which database goes down, the complementary service will take over. For safety purposes, the information will be saved in a queue located on the same machine as the sane database to avoid the loss of information in case of the second machine failure. All the servers are booted from the master machine, via the Bridge server provided by Tuxedo that uses the network connection and the *tlisten* process on both machines that listen on the same port. The Master/Backup role of the two machines (DBBL, BBL) can be switched between them via a script that monitors their sanity, avoiding the loss of the system in case of Master machine crash.

6. Implementation, Testing and Results

6.1. Application Configuration

A Tuxedo application is parameterized via an ASCII configuration file. This file describes the application using a set of parameters that the software interprets to create a run able application.

The main sections that need to be configured in the UBBCONFIG file are (see Appendix A - Configuration File):

- The RESOURCES section describes the global parameters for the application such as the Master machine, type of machine (MP), maximum number of servers, load balancing, system access etc.
- The MACHINES section contains the description of the machines used by the application. In our case there are two machines configured TUX1 and TUX2, where TUX2 is the master machine and TUX1 is the backup.
- The NETWORK section identifies the network addresses used by the administrative programs (e.g. listener and BBL).
- The GROUPS section describes the server groups that will execute on each machine. For our application we have defined eight groups, four on each machine that will manage the resources and server:
    - The groups ORA1 and ORA2 manage the two databases on each machine.
    - The groups QUE1 and QUE2 manage the two queues on each machine.
    - The groups APP_GRP1 and APP_GRP2 manage the servers that perform the

synchronous update of the resources on each machine.

- The groups APP_QUE1 and APP_QUE2 manage the servers that update the queues on each machine.

- The SERVER section lists the server names with their respective groups. Each entry in this section represents a server process to be booted in the application. Our servers are defined as follows:
  - *mainserv*: manages the global transactions that updates the two resources, located on the two machines.
  - *dbupdate* and *dbupdate2*: perform the update of the respective databases.
  - *qupdate* and *qupdate2*: perform the update of the respective queues.
  - *deqserv* and *deqserv2*: retrieve the messages from the respective queues.
- The SERVICES section contains parameters such as relative workload, data dependent routing etc.

The services advertised by the above-described servers are:

- CALLRM: manage the execution of the global transaction.
- UPDATE1: Insert the message into the Database 1.
- UPDATE2: Insert the message into the Database 2.
- QUPDATE1: Insert the message into the Queue 1.
- QUPDATE2: Insert the message into the Queue 2.
- DEQUE1: Extract the message from the Queue 1.
- DEQUE2: Extract the message from the Queue 2.
- UNDOUPDATE1: Advertise the UPDATE1 service under a different name.
- UNDOUPDATE2: Advertise the UPDATE2 service under a different name.

- REDOUPDATE1: Advertise again the UPDATE1 service.
- REDOUPDATE2: Advertise again the UPDATE2 service.
- UNDOQUPDATE1: Unadvertise the QUPDATE1 service.
- UNDOQUPDATE2: Unadvertise the QUPDATE2
- REDOQUPDATE1: Advertise again the QUPDATE1 service.
- REDOQUPDATE2: Advertise again the QUPDATE2 service.

The configuration file is then compiled to produce a binary version of this file (TUXCONFIG) that is copied on all participating machines along with the compiled servers and clients before the application can be booted.

To create distributed transactions processing, we must create a global transaction log (TLOG) on each participating machine. Parameters defined for TLOG should match the corresponding parameters defined in the Machines section of the UBBCONFIG file.

For a network application, we must start the listener process on each machine (e.g. *tlisten IPAddress: Port*). The port on which the process is listening must be the same as the port specified in the NETWORK section of the configuration file.

The application is normally booted using *tmboot* command, from the machine designated as the MASTER in the RESOURCES section of the configuration file.

6.2. Server and Client Applications

As specified before, all the server and the client programs are written using a C interface (ATMI) provide by Tuxedo. The servers and clients are compiled together with Tuxedo and started at the boot time.

Figure 13 shows the list of servers and services when the application is started. The servers are distributed over the two machines to insure the functionality of the application in case of database or node failure.

> psc

Service Name	Routine Name	Prog Name	Grp Name	ID	Machine	# Done	Status
TMS	TMS	TMS	APP_Q+	30001	TUX1	-	AVAIL
TMS	TMS	TMS	APP_Q+	30001	TUX2	-	AVAIL
TMS	TMS	TMS_ORA	ORA2	30001	TUX2	-	AVAIL
UPDATE1	UPDATE1	dbupdt	ORA1	1	TUX1	-	AVAIL
UNDOUPDATE1	UNDOUPDATE1	dbupdt	ORA1	1	TUX1	-	AVAIL
REDOUPDATE1	REDOUPDATE1	dbupdt	ORA1	1	TUX1	-	AVAIL
TMS	TMS	TMS_ORA	ORA1	30001	TUX1	-	AVAIL
TMS	TMS	TMS_QM	QUE2	30001	TUX2	-	AVAIL
TMS	TMS	TMS_QM	QUE1	30001	TUX1	-	AVAIL
TMS	TMS	TMS	APP_Q+	30002	TUX1	-	AVAIL
TMS	TMS	TMS	APP_Q+	30002	TUX2	-	AVAIL
UPDATE2	UPDATE2	dbupdt2	ORA2	2	TUX2	-	AVAIL
UNDOUPDATE2	UNDOUPDATE2	dbupdt2	ORA2	2	TUX2	-	AVAIL
REDOUPDATE2	REDOUPDATE2	dbupdt2	ORA2	2	TUX2	-	AVAIL
TMS	TMS	TMS_ORA	ORA2	30002	TUX2	-	AVAIL
TMS	TMS	TMS_ORA	ORA1	30002	TUX1	-	AVAIL
TMS	TMS	TMS_QM	QUE2	30002	TUX2	-	AVAIL
TMS	TMS	TMS_QM	QUE1	30002	TUX1	-	AVAIL
TMS	TMS	TMS	APP_Q+	30003	TUX1	-	AVAIL
TMS	TMS	TMS	APP_Q+	30003	TUX2	-	AVAIL
CALLRM	CALLRM	mainserv	APP_C+	5	TUX1	-	AVAIL
CALLRM	CALLRM	mainserv	APP_C+	6	TUX2	-	AVAIL
DEQUE1	DEQUE1	deqserv	APP_C+	7	TUX1	-	AVAIL
DEQUE2	DEQUE2	deqserv2	APP_C+	8	TUX2	-	AVAIL
UNDOQUPDATE1	UNDOQUPDATE1	queupdt	APP_Q+	11	TUX2	-	AVAIL
REDOQUPDATE1	REDOQUPDATE1	queupdt	APP_Q+	11	TUX2	-	AVAIL
QUPDATE1	QUPDATE1	queupdt	APP_Q+	11	TUX2	-	AVAIL
UNDOQUPDATE2	UNDOQUPDATE2	queupdt2	APP_Q+	12	TUX1	-	AVAIL
REDOQUPDATE2	REDOQUPDATE2	queupdt2	APP_Q+	12	TUX1	-	AVAIL
QUPDATE2	QUPDATE2	queupdt2	APP_Q+	12	TUX1	-	AVAIL
QSPACE	TMQUEUE	TMQUEUE	QUE1	20	TUX1	-	AVAIL
QSPACE2	TMQUEUE	TMQUEUE	QUE2	21	TUX2	-	AVAIL

Figure 13 List of Servers and services

From the functionality point a view there are two main servers: the server that updates the resources (*mainserv*), which runs on both machines, and the recovery server that is specific to each machine/queue (*deqserv, deqserv2*).

The following figures depict the sequence of events for a request sent by the client application (*maincl*) to update the databases. When a request is sent to the system there are four ways to handle this request depending on the availability of the system:

- Normal Mode (Figure 14): the Client (*maincl*) calls the CALLRM service (*mainserv*) which acts as a dispatcher. CALLRM calls both database resource services UPDATE1 (*dbupdt*) and UPDATE2 (*dbupdt2*). If both services are successful, then the transaction is successful and the Client receives the confirmation.

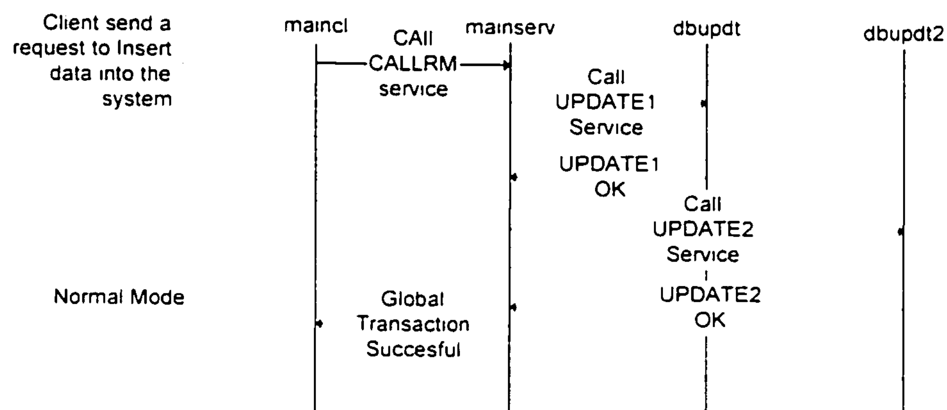

Figure 14 Sequence Diagram for Normal Mode

- Degraded Mode (Figure 15): If one of the databases is not available, hence UPDATE1 or UPDATE2 service returns a failure and CALLRM service calls QUPDATE1 (*qupdt*) or QUPDATE2 (*qupdt2*) services. If the Queue1 or Queue2 are updated successfully then the transaction succeeds and the Client is informed that the message has been inserted, without noticing the system partial failure.

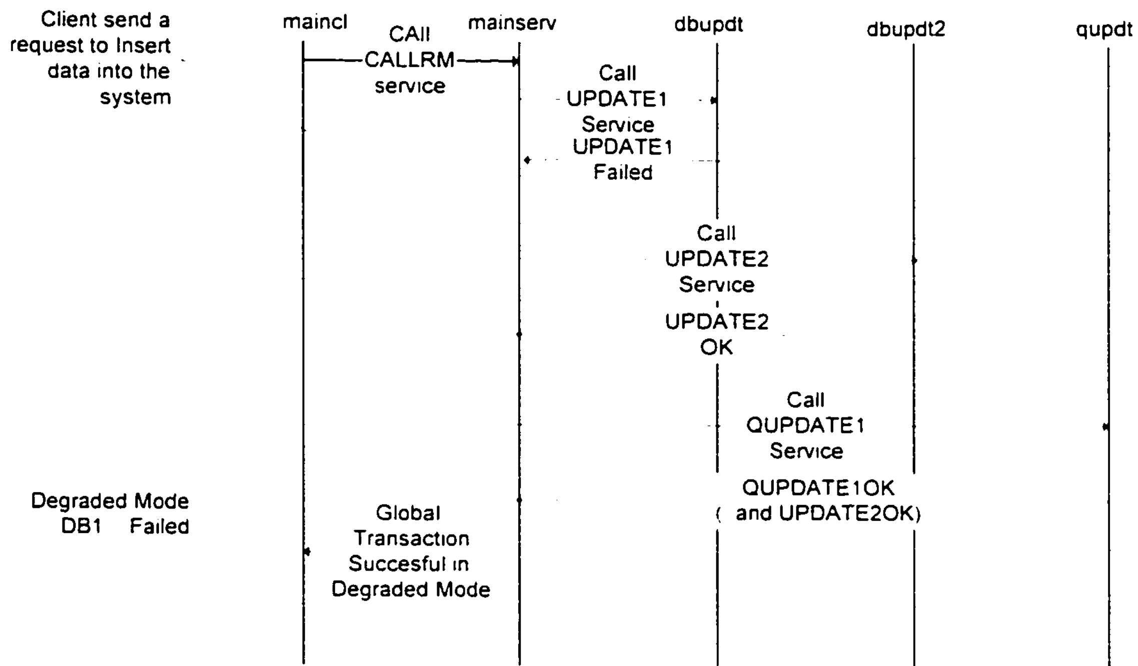

Figure 15 Sequence Diagram for Degraded Mode

- Failure Mode: The last case is the system's total failure when neither two databases nor one database and a queue can be updated simultaneously, and the transaction is rejected. To insure that when a database/machine goes down a queue is always available, the system will always try to update the queue located on the same machine with the available database (e.g. update DB1 and Queue 1 both located on TUX1).

When the system functions in the Degraded Mode and the failed database/node is recovered up to the time of the failure, the administrator has to launch a Client (*deqcl*) that will synchronize the two databases. Before starting this Client the Tuxedo administrator has to restart all the services on the failed node.

Figure 16 illustrates the sequence of events that takes place once the Recovery procedure is started. First the Client (*deqcl*) calls the UNDOUPDATE1 (*dbupdt*) service that makes the UPDATE1 service available only to the administrator until the

recovery process is done. Then the client calls the DEQUE (*qupdt*) service that extracts messages from the queue and then updates the database via DBUPDATE1 service (UPDATE1 with a different name). When the queue is empty then the recovery is complete.

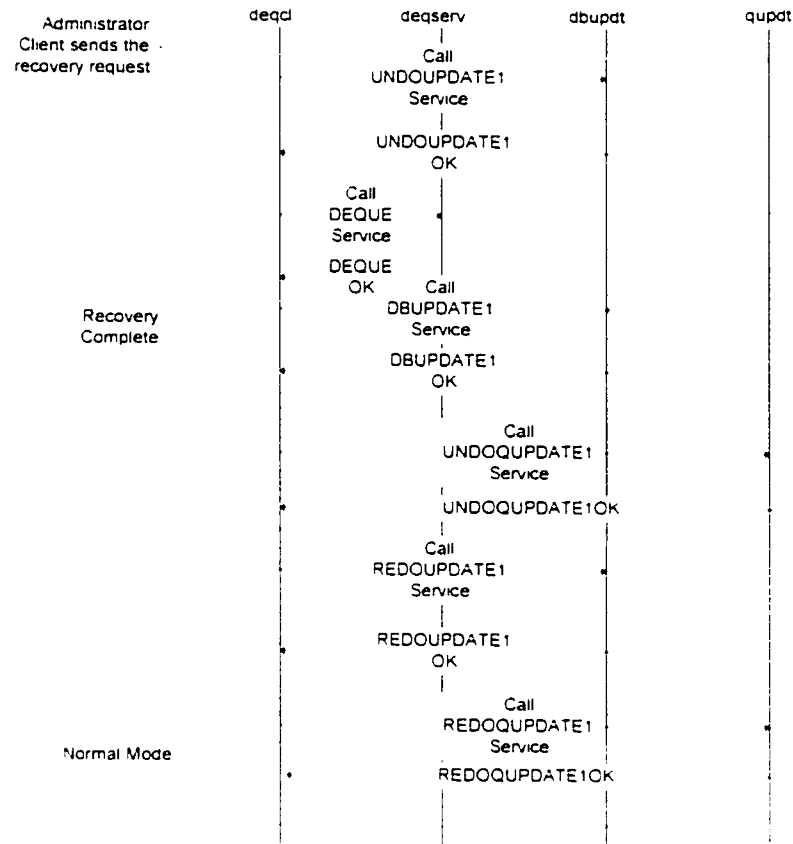

Figure 16 Sequence Diagram for the Recovery procedure

To make sure that the queue is not updated anymore once the queue is empty, the Client calls UNDOQUPDATE1 service that removes the QUPDATE1 from the list of available services and immediately after the REDOUPDATE1 service makes the UPDATE1 service available to get back to the Normal Mode.

Transactions may fail during this switch and to prevent their loss the regular Client (*maincl*) retries three times to commit a transaction before advertising the user that the transaction has failed. The last step is to restore the QUPDATE1 service by calling REDOQUPDATE1 service. At this moment the system is again in the Normal Mode having all the resources and services available.

Note 6-1: The reason that the services UPDATE1, UNDOUPDATE1 and REDOUPDATE1 belong to the same server (*dbupdt*) is that a service can be advertised or unadvertised only from within itself. This a Tuxedo limitation. The same principle applies to the other servers that deal with resources (e.g. *dbupdt2, qupdt, qupdt2).*

6.3. Administrative scripts

In case of failure there are two scenarios that need different actions from the administrator's side:

- Node failure (Figure 17): If a node fails and all the process running on that machine die including the remote BBL, then the DBBL will report the processes running on the failed machine as Partitioned. If the Master node fails the DBBL is also lost.

This means that the application will function using the servers that reside on the surviving node.

```
> psc
Service Name Routine Name Prog Name  Grp Name  ID      Machine  # Done Status
------------ ------------ ---------  --------  --      -------  ------ ------
TMS          TMS          TMS        APP_Q+  30001     TUX1          - PART
TMS          TMS          TMS        APP_Q+  30001     TUX2          - AVAIL
TMS          TMS          TMS_ORA    ORA2    30001     TUX2          - AVAIL
UPDATE1      UPDATE1      dbupdt     ORA1        1     TUX1          - PART
UNDOUPDATE1  UNDOUPDATE1  dbupdt     ORA1        1     TUX1          - PART
REDOUPDATE1  REDOUPDATE1  dbupdt     ORA1        1     TUX1          - PART
TMS          TMS          TMS_ORA    ORA1    30001     TUX1          - PART
TMS          TMS          TMS_QM     QUE2    30001     TUX2          - AVAIL
TMS          TMS          TMS_QM     QUE1    30001     TUX1          - PART
TMS          TMS          TMS        APP_Q+  30002     TUX1          - PART
TMS          TMS          TMS        APP_Q+  30002     TUX2          - AVAIL
UPDATE2      UPDATE2      dbupdt2    ORA2        2     TUX2          - AVAIL
UNDOUPDATE2  UNDOUPDATE2  dbupdt2    ORA2        2     TUX2          - AVAIL
REDOUPDATE2  REDOUPDATE2  dbupdt2    ORA2        2     TUX2          - AVAIL
TMS          TMS          TMS_ORA    ORA2    30002     TUX2          - AVAIL
TMS          TMS          TMS_ORA    ORA1    30002     TUX1          - PART
TMS          TMS          TMS_QM     QUE2    30002     TUX2          - AVAIL
TMS          TMS          TMS_QM     QUE1    30002     TUX1          - PART
TMS          TMS          TMS        APP_Q+  30003     TUX1          - PART
TMS          TMS          TMS        APP_Q+  30003     TUX2          - AVAIL
CALLRM       CALLRM       mainserv   APP_Q+      5     TUX1          - PART
CALLRM       CALLRM       mainserv   APP_Q+      6     TUX2          - AVAIL
DEQUE1       DEQUE1       deqserv    APP_Q+      7     TUX1          - PART
DEQUE2       DEQUE2       deqserv2   APP_Q+      8     TUX2          - AVAIL
UNDOQUPDATE1 UNDOQUPDATE1 queupdt    APP_Q+     11     TUX2          - AVAIL
REDOQUPDATE1 REDOQUPDATE1 queupdt    APP_Q+     11     TUX2          - AVAIL
QUPDATE1     QUPDATE1     queupdt    APP_Q+     11     TUX2          - AVAIL
UNDOQUPDATE2 UNDOQUPDATE2 queupdt2   APP_Q+     12     TUX1          - PART
REDOQUPDATE2 REDOQUPDATE2 queupdt2   APP_Q+     12     TUX1          - PART
QUPDATE2     QUPDATE2     queupdt2   APP_Q+     12     TUX1          - PART
QSPACE       TMQUEUE      TMQUEUE    QUE1       20     TUX1          - PART
QSPACE2      TMQUEUE      TMQUEUE    QUE2       21     TUX2          - AVAIL

>
```

Figure 17 Services status after machine failure

In case of Master node failure the Master role needs to be switched to the Backup machine and the entries of the dead servers and services cleaned from the DBBL (e.g. *pclean TUX1*). This process can be automated via a monitoring script that checks the sanity of the system (heartbeat) and when it detects a node failure it will execute the clean up script as the one below, on the remaining node:

```
tmadmin <<!
echo
master
pclean TUX1
q
!
```

The recovery process needs manual intervention due to the fact that the database has to be brought up to the point in time of the failure. However, this intervention is minimal and the downtime is transparent to the user, so it has no impact on the availability of system.

In order to synchronize the two databases without interrupting the service, the following steps are necessary: boot the application server on the failed node and run the database synchronization script. By using a "shell" scrip as the one below, the processes will be restarted and the databases synchronized while the system is running, without impacting its functionality:

```
tmboot -B TUX1
tmboot -l TUX1
/deqcl
```

- Database failure: If the database fails then the services related to this resource would be affected, but not the overall application. After that database has been restored up to the time of the failure the services belonging to that group have to be restarted and then start the actual recovery. This again can be done by using a "shell" script:

```
tmshutdown -g ORA1
tmboot -g ORA1
/deqcl
```

When these scripts have finished execution the system is back in the Normal Mode; however, throughout this recovery process the application is running and its functionality is not impacted.

6.4. Experiment and results

The goal of the testing is to verify the behavior of the system in case of software and hardware failures testing that the database recovery mechanism could be performed without transaction loss. The tests consists of failures in database, server processes, Master node, Backup node and a packet of tests for measuring the systems response time.

Test Name	Description	Results
Database Failure	One of the databases is shutdown ungracefully (*shutdown abort*) to simulate a database failure.	Passed (See Note 6-2)
Server Process Failure	Kill multiple server processes on one machine.	Passed (See Note 6-3)
Network Failure	Shutdown Tuxedo's listener (kill "*tlistener*" process).	Passed (See Note 6-4)
Master Node Failure	The Master machine is rebooted while the system is running.	Passed (See Note 6-5)
Backup Node Failure	The Backup machine is rebooted while the system is running.	Passed (See Note 6-6)
Response time	Measure the number of transactions per second for: a) update of the two databases b) update one database and one queue c) read from queue and update one database	Passed (See Note 6-7)

Table 6-1: Test Overview

Note 6-2: Database Failure

The test was performed by shutting down the Database 1 completely and/or by shutting down the database listener. Both machines were tested and the system handled well the loss of one database. The transaction that is executing at the time of the switch is rolled back and handled by the retry mechanism introducing a 2 seconds delay. Tuxedo Application Server sees a database as a resource, and if a resource is not available, the system is automatically switched to Degraded Mode where a queue is updated. Depending on the number of the transactions/second, the queue can be designed to handle hours of Degraded Mode. The development system can handle maximum 55 transactions/sec. hence for a Degraded Mode of for example two hours, the queue has to be designed for at least 400.000 messages. The administrator has to make sure that enough disk space is allocated for the queues, to secure the transactions during the degraded mode period.

Note 6-3: Server Process Failure

When server processes are killed randomly on both machines, the system is restarting them automatically after they are declared dead. This may introduce a 2 seconds delay in treating the current transaction that is rolled back and retried. The transaction that is executing when the processes are killed, it is rolled back but succeeds at retry using the new server process.

Note 6-4: Network Failure

Shutting down the Tuxedo's listener, simulating the network failure performed the test. In this case, there is no communication between the services on both machines; hence

from a client point of view the second database is not available. The recovery scenario is the same as for the database failure test, the queue being accessed locally to avoid any data loss.

Note 6-5: Master Node failure

The test was performed by shutting down the Master Node. In this case the system loses the DBBL that monitors all the servers the overall sanity of the system.

The Master node failure is the most critical test, having as a result the loss of all the processes that run on this machine and of the DBBL. The Backup node it is aware that the servers on the Master node are not available and the application is switched to Degraded Mode. This way the system continues to function in Degraded Mode without service interruption. In the recovery process the Backup node needs to be switched to Master in order to create a new DBBL then the failing node can be booted as Backup.

Note 6-6: Backup Node failure

The test was performed by shutting down the Backup Node. In this case the system functions in Degraded Mode making use of the services that run on the remaining node. The Backup node failure has as a result the loss of all the processes that run on this machine. Tuxedo's DBBL reports this loss as a "Partitioning" of the system. The only services that will remain unavailable are the ones related to the database that is physically located on the Backup machine. Therefore, the processing continues in Degraded Mode until the recovery process can be performed, but no transactions are lost.

Figure 18 and Figure 19 illustrate a node failure and the recovery process. Figure 18 shows that while the client is running trying to insert a batch of 1000 messages, one of the nodes it is rebooted. The system is switched automatically to a Degraded Mode, wherein the queue (QSPACE2) is updated. By listing the messages in the queue we see that the messages were properly stored. While we performed this tested we simulated that the clients continue running hence another 1000 messages are stored in the queue.

```
Terminal
Window  Edit  Options

Inserted string is: test2   259 times
Inserted string is: test2   260 times
Inserted string is: test2   261 times
Inserted string is: test2   262 times
Inserted string is: test2   263 times
Inserted string is: test2   264 times
Inserted string is: test2   265 times
Inserted string is: test2   266 times
Inserted string is: test2   267 times
Inserted string is: test2   268 times
Inserted string is: test2   269 times
Inserted string is: test2   270 times
Inserted string is: test2   271 times
Inserted string is: test2   272 times
Inserted string is: test2   273 times
Inserted string is: test2   274 times
Inserted string is: test2   275 times
Inserted string is: test2   276 times
Inserted string is: test2   277 times
Inserted string is: test2   278 times
Inserted string is: test2   279 times
Inserted string is: test2   280 times
Inserted string is: test2   281 times
Inserted string is: test2   282 times
Inserted string is: test2   283 times
Inserted string is: test2   284 times
```

```
Terminal
Window  Edit  Options                                   Help

tuxedo   31383  3157  0 22:43 pts/0    00:00:00 ps -fu
[tuxedo@yul-tux-lnx1 simpapp]$ su
Password:
[root@yul-tux-lnx1 simpapp]# reboot -f

qmadmin - Copyright (c) 1996-1999 BEA Systems, Inc.
Portions * Copyright 1986-1997 RSA Data Security, Inc.
All Rights Reserved.
Distributed under license by BEA Systems, Inc.
Tuxedo is a registered trademark.
QMCONFIG=/home/tuxedo/simpapp/QUE1

> qopen QSPACE2

> qinfo
Queue JOURNAL1: 1715 Persistent Messages using 1715 pages.
                0 Nonpersistent Messages using 0 bytes.
                1715 Messages in total using 878080 bytes.
Queue ERRORQ: 0 Persistent Messages using 0 pages.
              0 Nonpersistent Messages using 0 bytes.
              0 Messages in total using 0 bytes.
```

Figure 18 Node Failover Demo

In Figure 19 we illustrate the recovery process where the failed node is booted, therefore made available to the application and the messages dequeued and inserted into the recovered database, while another client is running. The transactions that occur while the synchronization process is running continue to be logged into the queue, until the time of the switch when the transactions are logged again into the two databases.

```
exec TMS_QM -A :
        on TUX1 -> process id=1519 ... Started.
exec TMS_QM -A :
        on TUX1 -> process id=1520 ... Started.
exec TMQUEUE -s QSPACE:TMQUEUE -- :
        on TUX1 -> process id=1521 ... Started.
exec TMS_ORA -A :
        on TUX1 -> process id=1522 ... Started.
exec TMS_ORA -A :
        on TUX1 -> process id=1525 ... Started.
exec dbupdt -A :
        on TUX1 -> process id=1528 ... Started.
exec TMS -A :
        on TUX1 -> process id=1531 ... Started.
exec TMS -A :
        on TUX1 -> process id=1532 ... Started.
exec TMS -A :
        on TUX1 -> process id=1533 ... Started.
exec queupdt2 -A :
        on TUX1 -> process id=1534 ... Started.
10 processes started.

> q
[tuxedo@yul-tux-lnx2 simpapp]$ ./deqcl
Dequeued   1715 messages
[tuxedo@yul-tux-lnx2 simpapp]$
```

```
Inserted string is: test3   975 times
Inserted string is: test3   976 times
Inserted string is: test3   977 times
Inserted string is: test3   978 times
Inserted string is: test3   979 times
Inserted string is: test3   980 times
Inserted string is: test3   981 times
Inserted string is: test3   982 times
Inserted string is: test3   983 times
Inserted string is: test3   984 times
Inserted string is: test3   985 times
Inserted string is: test3   986 times
Inserted string is: test3   987 times
Inserted string is: test3   988 times
Inserted string is: test3   989 times
Inserted string is: test3   990 times
Inserted string is: test3   991 times
Inserted string is: test3   992 times
Inserted string is: test3   993 times
Inserted string is: test3   994 times
Inserted string is: test3   995 times
Inserted string is: test3   996 times
Inserted string is: test3   997 times
Inserted string is: test3   998 times
Inserted string is: test3   999 times
Inserted string is: test3  1000 times
[tuxedo@yul-tux-lnx1 simpapp]$
```

Figure 19 Node Fallback Demo

The number of messages in the queue is the same as the number of messages that are recovered (1715 messages), hence the two databases are now synchronized. Subsequently the system reverts to a normal mode wherein both databases are updated.

Note 6-7: Response time measurements

The test was performed with a batch of 10.000 messages using a special client application (*addcl*):

a) *Normal Mode*: update of the two databases : ~ 3 minutes

b) *Degraded Mode*: update one database and one queue: ~ 2.5 minutes

c) *Recovery Mode*: read from queue and update one database: ~ 1.5 minutes

The throughput of the system in Normal Mode it is almost 55 transactions sec.

The overall test results bring us to the conclusion that the prototype can handle the failures and most important can make this failures transparent to the users. These tests were

chosen and performed to demonstrate the robustness and availability of the system in case of major failures like node or database failure.

The results show that the system functions in Degraded Mode in case of failure. The Degraded Mode does not influence the performance of the system and it is totally transparent to the users. The recovery phase is also done in parallel with the regular transaction processing: it doesn't cause downtime or performance decrease. The way the recovery is handled reduces the manual intervention: the administrator has to run one script to synchronize the two databases, hence the risk of human error is greatly reduced.

These tests demonstrate that the prototype can handle graciously the failover and the fallback. However for a real life solution the tests have to be extended to real life scenario such as: testing the performance of the systems for large number of concurrent users/connections and also the database modifications serialization etc.

Considering that even in the worst case scenario, when the system needs to be replaced and the application rebooted, it does not take more than 3 minutes for the system to be back on-line. This indicates that the high availability criteria are met. Though the system was not tested with real traffic to demonstrate 99.999 % availability nonetheless, by making the outages transparent to the users and being able to do the database synchronization from the write-ahead-log(queue) without service interruption or database inconsistency, the prototype demonstrates that this is a viable solution. The solution can be evolved towards a system that can ensure the high availability necessary for a real time system.

7. Conclusion and future work

The proposed solution presented in this project is a response to the problem of providing a highly available system for a transactional system using databases. In our case Tuxedo. introduces a new level of complexity in the design phase and administration. but on the other hand the database is relieved from all the responsibilities except as a repository. This makes the recovery a lot easier in case of crash. Tuxedo also adds an extra cost to the system on top of the cost of Oracle or other RDBMS. But compared to the cost of hardware and software necessary for example to purchase a clustered database (e.g. Oracle's Real Cluster Application) it is well worth it.

The project focuses more on the recovery mechanism and the database synchronization with no service loss. There are certain aspects that the project does not tackle. like different types of operations that can occur in a database (e.g. Updates. Deletes. Schema modifications etc). This system is a solution for a transactional environment where all the messages coming from the outside world are logged in a database and once stored. the data is not modified (this being the typical use for recording events e.g. telecommunication world).

Real Time replication of data is not ideal because it adds extra constraints in our case the delay introduced by the synchronous database update. However. it is the only way to make sure that the data is synchronized so when an application fails a switchover from one node to another is instantaneous: the users see no service interruption. To enhance the performance. we can use an asynchronous update of the resources within the same transaction. having as result. a faster access to various resources (e.g. if more that two

databases are updated). In this case the response has to be handled via programming as opposed to the synchronous update that is handled by the dual-commit protocol.

One extra security measure that can be taken is to have a third database. which can be a Standby copy of the Backup database. hence update after event. This database/machine can be used as spare in case of failure to reduce the time the database is brought up to the point in time of failure.

The main effort is required by the application configuration on Tuxedo's side that requires good analysis skills in a distributed environment. The programming effort is not extensive. being reduced to a number of C routines that make use of Tuxedo's ATMI library.

The proof of concept presented in this project makes use of minimum capabilities that Tuxedo offers. To apply this architecture to a real life application. there are several features that could be added to the proposed design. to handle large volume of data. First the load balancing option that allows the configuration of the percentage of load for each service and group while allowing the remaining to be routed to alternate resources. For example if a queue is overloaded the traffic is automatically routed to the alternate server that has a queue with the same name. (e.g. 30% by Machine 1 and 70% by Machine 2). The system can be configured to start multiple instances of the same server to accommodate the extra load. Certain services can have priority over others. allowing the system to handle messages in a different manner. This is also configurable on a service/group basis and can also be the object of future development.

In the real world. the Client applications are not local. they being located on various workstations. These clients are managed by Tuxedo. and in case of failure. routed to

alternate nodes. This feature was not explored in this project, and can be the object of future improvements. Also the security aspect of such a system was not at all taken into account. Tuxedo offers an authentication mechanism that can be added to the design of the Workstation Clients.

Also as future work the application can be evolved to handle heterogeneous databases. This is easily feasible by using a different Resource Manager provided by the database manufacturer. In order to speed up the failover process in a production system a third party software should be used to monitor the sanity of the system. For example Lifekeeper from SteelEye is a very good choice [9] and in our case it can be used to advise the Backup system when the Master node is not available.

In a distributed environment, the management of different resources is made easier by expanding the client-server model to a distributed. Another feature that is very useful in a real-life scenario is the data-dependent routing. Thus, different databases can be used to store the data, depending on the value to be recorded. Through this feature, the scalability of the system can be enhanced without diminishing the performance.

Therefore, we can say that the proposed system presented offers an interesting solution to the database high availability issue by making use of a middleware, Tuxedo in our case, to solve the synchronization and the recovery issues.

8. References

[1] Clustra Systems Availability Analysis –When "five nines" is not enough – 2001 Clustra Systems Inc.

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/Clustra_Availability_WP.pdf

[2] The five 9s Pilgrimage: Toward the Next Generation of High-Availability Systems –2000 Clustra Systems Inc.

Archived in : http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/Clustra_5Nines_WP.pdf

[3] Raid Solutions - 2000, 2001 Advanced Computer & Network Corporation

URL: http://www.acnc.com/04_01_00.html

[4] DB2 and High Availability on SUN Cluster 3.0- IBM Corporation- August 2001

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/suncluster.pdf

[5] Compaq Corporation - Open VMS –Clusters Overview, May 1, 1998.

URL:http://www.openvms.compaq.com/openvms/products/clusters/Clusters_Overview.html

[6] HP Cluster Server Solutions Overview, 2002 HP Corporation

URL: http://www.hp.com/products1/storage/disk_arrays/xpstoragesw/cluster/index.html

[7] Polyserve: White Paper –Data Replication for High Availability Web Server Clusters

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/datarep_highavail.pdf

[8] Veritas Cluster Server V2.0 - Technical Overview

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/vcs20_techover_final_0901.pdf

[9] LifeKeeper for Linux – 2001 SteelEye technology Inc.

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/kpr4linux.pdf

[10] OPEN SANs – An In-Depth Brief, IBM White Paper, December 2000

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/open_san.pdf

[11] Polyserve: White Paper –Implementing Highly Available Commercial Systems under Linux using Data Replication –Dec. 2000 –Doug Delay

Archived in: http://www.cs.concordia.ca/~faculty/bcdesai/grads/stellab/data_rep_wp.pdf

[12] Implementing an Automated Standby Database –by Rob Sherman – May 1999, Oracle Magazine

URL: http: www.oracle.com oramag oracle/99-May index.html?39or8i.html

[13] Creating Hot Snapshots and Standby Databases with IBM DB2 Universal Database and V7.2 and EMC TimeFinder. IBM 2001

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab timefinder.pdf

[14] Linux and High-Availability Computing. Clustering Solutions for e-Business Today- Intel Corporation

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab linux_highavail.pdf

[15] Comparing Replication Technologies. PeerDirect Inc 2002

URL: http: www.peerdirect.com Products WhitePapers ComparingTechs.asp

[16] Implementing Highly Available Commercial Application under Linux using Data Replication -Doug Delaney -December 21, 2000. Polyserve. Inc

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab data_rep_wp.pdf

[17] Clustra Database 4.1 – Concepts Guide. Replication Architecture. July 2001 PeerDirect Inc.

URL: http: www.peerdirect.com Products WhitePapers Architecture.asp

[18] Oracle 8i Parallel Server Concepts Release 2 (8.1.6). Oracle Corporation 2001

[19] Extended Parallel Option for Informix Dynamic Server For Windows NT and UNIX -2001 Informix Corporation.

[20] Clustra Database - Technical Overview – 2000 Clustra Systems Inc.

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab Clustra_Technical_WP.pdf

[21] Building Highly Available Database Servers Using Oracle Real Application Clusters –An Oracle White Paper. May2001

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab high_avail_clusters.pdf

[22] Oracle 8I Server –Distributed Database Systems- Release 2 (8.1.6) – December 1999 Oracle Corporation

[23] Oracle 9I Real Application Cluster- Cache Fusion Delivers Scalability–An Oracle White Paper. May2001.

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab appclusters_cache.pdf

[24] Global Transactions –X Open XA – Resource Managers - Donald A. Marsh. Jr.- January 2000 Aurora Information Systems Inc.

URL: http: www.aurorainfo.com wp3

[25] Distributed Transaction Processing– The XA– Specification. Version 2 – June 1994 The Open Group

[26] Encina Transactional Programming Guide –IBM Corporation 1997

URL:http:/ www.transarc.ibm.com Library documentation txseries 4.2 aix en_US html aetgpt aetgpt02.htm#ToC_182

[27] Global Transactions in BEA TUXEDO System. BEA Corporation

URL: http: edocs.beasys.com tuxedo tux65 proggd tpg05.htm#997149

[28] XA-Server Integration Guide for TUXEDO – 1997 Sybase Inc.

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab xatuxedo.pdf

[29] IBM CICS Family: General Information-Copyright International Business Machines Corporation 1982, 1995. All rights reserved. Document Number: GC33-0155-05

URL: http: www-4.ibm.com software ts cics about

[30] BEA TUXEDO –The programming model -White Paper - November 1996

URL: http: www.bea.com products tuxedo paper_model.shtml

[31] High Availability Features - 2001 Oracle Corporation.

URL: http: technet.oracle.com deploy availability htdocs ha_features.html

[32] Issues Insufficiently Resolved in Century 20 in the Fault Tolerant Distributed Computing Field – Kane Kim UCI Dream Lab- Oct 2000

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab srds2000_slides.pdf

[33] Achieving Continuos Availability with Sybase Adaptive Server –2001 Sybase Inc.

URL: Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab HASybasePaper6.pdf

[34] Informix Enterprise Replication a High performance solution for Distributing and Sharing Information 1998 Informix Corporation

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab entrep.pdf

[35] High Availability through Warm-Standby Support in Sybase Replication Server –1998 Sybase Inc.

Archived in: http: www.cs.concordia.ca ~faculty bcdesai grads stellab kpr4linux.pdf warm_standby_wp.pdf

[36] Data Guard Concepts – 2001 Oracle Corporation.

URL: http: download-east.oracle.com otndoc oracle9i 901_doc server.901 a88808 standbycon.htm#57314

[37] The Stanford Rapide Project- X OPEN Architecture Rapide Code

URL:http: pavg.stanford.edu rapide examples xopen code.html

[38] The TUXEDO System –Software for Constructing and Managing Distributed Business Applications – Juan M. Andrade, Mark T. Carges, Terence J. Dwyer, Stephen D. Felts – November, 1997, also URL: http: www.bea.com products tuxedo paper model4.shtml

[39] H. Tai and K. Kosaka. *The Aglets Project.* Communications of the ACM, Vol. 42(3) pp.100-- 101, March 1999

[40] Robert Breton. Replication Strategies for High Availability and Disaster Recovery. Data Engineering Vol. 21(4) pp. 38-43

[41] Rosana S.G. Lanzelotte, Patrick Valduriez Mohamed Zait, Mikal Ziane, Industrial-Strength Parallel Query Optimization: issues and lessons. An International Journal, 1994, also http: poleia.lip6.fr ~ziane infsys.ps.gz

[42] M. Tamer Özsu, Patrick Valduriez, Distributed and Parallel Database Systems, ACM Computing Surveys, vol.28, no.1, pp 125-128, March 1996.

[43] L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems. Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, Taipe, Taiwan, April, 2000.

9. Appendices

9.1. Appendix A - Configuration File

```
*RESOURCES
IPCKEY          123456
DOMAINID        simpapp
MASTER          TUX2,TUX1
MAXACCESSERS    100
MAXSERVERS      100
MAXSERVICES     100
MODEL           MP
OPTIONS         LAN,MIGRATE
LDBAL           N
SYSTEM_ACCESS   PROTECTED
SCANUNIT  5
SANITYSCAN      2
DBBLWAIT  5
BBLQUERY  2
BLOCKTIME 20
*MACHINES
DEFAULT:
                APPDIR="/home/tuxedo/simpapp"
```

```
TUXCONFIG="/home/tuxedo/simpapp/tuxconfig"

TUXDIR="/opt/tuxedo8.0"

ENVFILE="/home/tuxedo/simpapp/ubb.env"


"yul-tux-lnx1"   LMID=TUX1

        TLOGDEVICE="/home/tuxedo/simpapp/TLOG"

        TLOGNAME=TLOG

        TLOGSIZE=100


"yul-tux-lnx2"   LMID=TUX2

            TLOGDEVICE="/home/tuxedo/simpapp/TLOG"

            TLOGNAME=TLOG

            TLOGSIZE=100


*NETWORK

TUX1    NADDR="57.4.164.47:3334"

    NLSADDR="57.4.164.47:2334"

TUX2    NADDR="57.4.164.48:3334"

    NLSADDR="57.4.164.48:2334"


*GROUPS

ORA1

      LMID=TUX1
```

```
        TMSNAME=TMS_ORA
        GRPNO=1
OPENINFO="Oracle_XA:Oracle_XA+Acc=P/adams/adamsboy+SqlNet=adamsdev
3+SesTm=60+LogDir=/home/tuxedo/simpapp"
        TMSCOUNT=2

ORA2
      LMID=TUX2
      TMSNAME=TMS_ORA
      GRPNO=2
OPENINFO="Oracle_XA:Oracle_XA+Acc=P/adams/adamsboy+SqlNet=adamsdev
4+SesTm=60+LogDir=/home/tuxedo/simpapp"
      TMSCOUNT=2

APP_GRP1
      LMID=TUX1
      GRPNO=3

APP_GRP2
      LMID=TUX2
      GRPNO=4

QUE1
```

LMID = TUX1

GRPNO = 5

TMSNAME = TMS_QM TMSCOUNT = 2

OPENINFO = "TUXEDO/QM:/home/tuxedo/simpapp/QUE1:QSPACE"

QUE2

LMID = TUX2

GRPNO = 6

TMSNAME = TMS_QM TMSCOUNT = 2

OPENINFO = "TUXEDO/QM:/home/tuxedo/simpapp/QUE1:QSPACE2"

APP_QUE1

LMID=TUX1

GRPNO=7

TMSNAME=TMS

APP_QUE2

LMID=TUX2

GRPNO=8

TMSNAME=TMS

*SERVERS

TMQUEUE

SRVGRP = QUE1 SRVID = 20

GRACE = 0 RESTART = Y CONV = N MAXGEN=10

CLOPT = "-s QSPACE:TMQUEUE --"

TMQUEUE

SRVGRP = QUE2 SRVID = 21

GRACE = 0 RESTART = Y CONV = N MAXGEN=10

CLOPT = "-s QSPACE2:TMQUEUE --"

dbupdt   SRVGRP= ORA1 SRVID=1 CLOPT="-A" GRACE = 0 RESTART = Y CONV = N MAXGEN=10

dbupdt2   SRVGRP= ORA2 SRVID=2 CLOPT="-A" GRACE = 0 RESTART = Y CONV = N MAXGEN=10

mainserv   SRVGRP= APP_GRP1 SRVID=5 CLOPT="-A" GRACE = 0 RESTART = Y CONV = N MAXGEN=10

mainserv SRVGRP= APP_GRP2 SRVID=6 CLOPT="-A" GRACE = 0

RESTART = Y CONV = N MAXGEN=10

deqserv SRVGRP= APP_GRP1 SRVID=7 CLOPT="-A" GRACE = 0

RESTART = Y CONV = N MAXGEN=10

deqserv2 SRVGRP= APP_GRP2 SRVID=8 CLOPT="-A" GRACE = 0

RESTART = Y CONV = N MAXGEN=10

queupdt SRVGRP= APP_QUE2 SRVID=11 CLOPT="-A" GRACE = 0

RESTART = Y CONV = N MAXGEN=10

queupdt2 SRVGRP= APP_QUE1 SRVID=12 CLOPT="-A" GRACE = 0

RESTART = Y CONV = N MAXGEN=10

*SERVICES

CALLRM AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

CALLDBQ AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

UPDATE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

UPDATE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

QUPDATE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

QUPDATE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

DEQUE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

DEQUE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

UNDOUPDATE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

UNDOUPDATE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

REDOUPDATE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

REDOUPDATE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

UNDOQUPDATE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

UNDOQUPDATE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

REDOQUPDATE1 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

REDOQUPDATE2 AUTOTRAN=Y TRANTIME=30 SVCTIMEOUT=10

9.2. Appendix B - Makefile

```
ORACLE_HOME=/app/oracle/product/8.1.7

PROC = $(ORACLE_HOME)/bin/proc

IN = iname

OUT = outname

IC =

/opt/tuxedo8.0/include,/app/oracle/product/8.1.7/precomp/public,/app/oracle/product
/8.1.7/rdbms/demo,/app/oracle/product/8.1.7/network/public,/app/oracle/product/8.1.
7/plsql/public

INCLUDES = $(IC)/atmi.h $(IC)/tx.h $(IC)/sqlcode.h $(IC)/userlog.h
```

```
all:mainserv dbupdt dbupdt2 queupdt queupdt2 deqserv deqserv2 deqcl deqcl2
maincl addcl tuxconfig

mainserv:mainserv.c
	buildserver -o mainserv -f mainserv.c -s CALLRM

deqserv:deqserv.c
	buildserver -o deqserv -f deqserv.c -s DEQUE1 -r TUXEDO/QM

deqserv2:deqserv2.c
	buildserver -o deqserv2 -f deqserv2.c -s DEQUE2 -r TUXEDO/QM

queupdt:queupdt.c
		buildserver -o queupdt -f queupdt.c -s QUPDATE1 -s
	REDOQUPDATE1 -s UNDOQUPDATE1

	queupdt2:queupdt2.c
		buildserver -o queupdt2 -f queupdt2.c -s QUPDATE2 -s
	REDOQUPDATE2 -s UNDOQUPDATE2

	dbupdt:dbupdt.pc
		$(PROC) iname=dbupdt.pc oname=dbupdt.c include=$(IC) \
```

```
	sys_include=($ORACLE_HOME/precomp/public,/usr/lib/gcc-
lib/i386-redhat-linux/egcs-2.91.66/include,/usr/include) code=ansi_c
	buildserver -o dbupdt -f dbupdt.c -s UPDATE1 -s UNDOUPDATE1 -
s REDOUPDATE1 -r Oracle_XA

dbupdt2:dbupdt2.pc
	$(PROC) iname=dbupdt2.pc oname=dbupdt2.c include=$(IC) \
	sys_include=($ORACLE_HOME/precomp/public,/usr/lib/gcc-
lib/i386-redhat-linux/egcs-2.91.66/include,/usr/include) code=ansi_c
	buildserver -o dbupdt2 -f dbupdt2.c -s UPDATE2 -s
UNDOUPDATE2 -s REDOUPDATE2 -r Oracle_XA

maincl:maincl.c
	buildclient -o maincl -f maincl.c

addcl:addcl.c
	buildclient -o addcl -f addcl.c

deqcl:deqcl.c
	buildclient -o deqcl -f deqcl.c

deqcl2:deqcl2.c
	buildclient -o deqcl2 -f deqcl2.c
```

tuxconfig:ubbconfig

tmloadcf -y ubbconfig

9.3. Appendix C - Environment Files

- **ubb.env file**

```
ORACLE_BASE=/app/oracle
ORACLE_HOME=/app/oracle/product/8.1.7
ORACLE_SID=adamsdev
TNS_ADMIN=/app/oracle/product/8.1.7/network/admin
RM=Oracle_XA
RMNAME=Oracle_XA
ORACLE_LIBS=/app/oracle/product/8.1.7/lib
ORACLE_DIR=/app/oracle/product/8.1.7/bin
```

- **tux.env file**

```
TUXDIR=/opt/tuxedo8.0: export TUXDIR
PATH=$TUXDIR/bin:$PATH: export PATH
LIBPATH=$TUXDIR/lib:$LIBPATH: export LIBPATH
LD_LIBRARY_PATH=$TUXDIR/lib:$LD_LIBRARY_PATH: export
LD_LIBRARY_PATH
WEBJAVADIR=$TUXDIR/udataobj/webgui/java
TUXCONFIG=/home/tuxedo/simpapp/tuxconfig: export TUXCONFIG
LANG=english_us.ascii:export LANG
ORACLE_BASE=/app/oracle: export ORACLE_BASE
```

ORACLE_HOME=$ORACLE_BASE/product/8.1.7:export ORACLE_HOME

ORACLE_SID=adamsdev: export ORACLE_SID

LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME/lib: export LD_LIBRARY_PATH

PATH=$PATH:$ORACLE_HOME/bin: export PATH

TNS_ADMIN=$ORACLE_HOME/network/admin: export TNS_ADMIN

RM=Oracle_XA:export RM

RMNAME=$RM:export RMNAME

ORACLE_LIBS=$ORACLE_HOME/lib:export ORACLE_LIB

ORACLE_DIR=$ORACLE_HOME/bin:export ORACLE_DIR

TMTRACE=on: export TMTRACE

QMCONFIG=/home/tuxedo/simpapp/QUE1: export QMCONFIG

9.4. Appendix D - Glossary

API – Application Programming Interface is the definition of the calling formats to a set of procedures that perform a set of related tasks.

Asynchronous Replication – After event update of a copy database.

Asynchronous Communications – communications in which the requestor does not wait for the response but will poll for it at a later time.

ATMI – Application to Transaction Monitor Interface, is the TUXEDO communications application-programming interface.

BBL - Bulletin Board Liaison is an administrative server on each computer in the application that manages application server.

BRIDGE - an administrative server that manages network communications between computers within an application.

Bulletin Board – a distributed, partially replicated, memory data structure used to maintain information in the TUXEDO system for name serving, transaction management, and run-time information.

Client – a software module that gathers and presents data to an application; it generates requests for services and receives replies.

DBBL - Distinguished Bulletin Board Liaison is an administrative server that runs on the master machine and manages the distribution of the Bulletin Board and global updates thereto.

DSA – Dynamic Scalable Architecture

DTPM – Distributed Transaction Processing Model.

Failover – The procedure that takes over the system in case of failure transferring the processing from the failed component to the backup component.

Fallback – The procedure that tries to restore the failed component and to bring it back on-line.

LAN - Local Area Network

Load Balancing – assignment of request to queues for servers offering the requested service in a manner to optimize throughput and/or response time.

Local Client – a client that runs on a computer where the TUXEDO System administrative programs an d application servers run.

Journal – raw file used to store the data in case of one database not being available.

MTTR – Mean Time to Recover. represents the time for a system to recover after failure.

MTBF – Mean Time between Failures. represents the time between two system failures.

OLTP- Online Transaction Processing.

OPS – Oracle Parallel Server.

Priority – processing of the messages based on an urgency factor assigned by the application.

Queue – a memory data structure to hold messages for processing.

/Q - the TUXEDO subsystem that provides reliable. application queues.

RAID – Redundant Arrays of Inexpensive Disks.

RDBMS - Relational Database Management System.

RM - Resource Manager a module or collection of modules. the most common of which are databases. that maintain the state of the application.

SAN- Storage Area Network. which are private networks for storage.

Server – a software module that accepts requests from the clients an other servers and returns replies.

Server Clusters – Servers grouped together that appear as a single server.

Server group – a collection of server that can be activated, deactivated and migrated as a unit.

Service – the name given to an application routine available for requests by a client in the system with well-defined inputs, outputs and processing.

Synchronous Communications – communications in which the requestor waits for the reply.

Synchronous Replication – Redundant write to two database systems.

TCP/IP- Transmission Control Protocol a standard operating system interface to networking services.

Time-out – an event that occurs when processing takes longer than expected or configured; time-outs can occur either when blocking for an operation or due to a transaction taking longer than specified by application.

TLOG – Transaction Log is a stable storage area where the completion of a transaction is logged.

tlisten – TUXEDO Listener – an administrative program that povides an entry point to all computers within the application; it is used primarily to propagate files and activate servers.

TM – Transaction Manager is the software that manages global transactions across multiple computers and resource managers.

TMS – Transaction Manager Server is an administrative server that manages the two-phase commit protocol and recovery for global transactions.

TUXCONFIG – the TUXEDO binary configuration file for an application definition.

Two-phase commit (2PC) – algorithm to ensure the atomicity and consistency of a global transaction which requires the update of a number of replicates at different nodes of a distributed database.

TX – an X/OPN interface for transaction demarcation: based on ATMI.

UBBCONFIG – the ASCII file where TUXEDO administrators can define and application configuration.

UPS - Uninterruptible Power Supplies.

Workstation Client - a client that accesses the TUXEDO System via the network.

X/OPEN – Standard body that developed the Distributed Transaction Processing Model.

XA- Interface that describes the protocol for transaction coordination, commitment and recovery between a Transaction manager and one or more resource Managers.