NTNU
Innovation and Creativity

Derby: Replication and Availability

Egil Sørensen

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In addition to the physical log in Derby, write a logical log to a Neighbor Node which acts as a hot standby for the database, ensuring replication and improving the availability of Derby. The hot standby database will redo the logical log records received. If the primary Derby database crashes the Hot Standby will become primary and accept connections. Finally, when the crashed Derby recovers it will synchronize itself with the now active primary and then reenter hot standby mode.

Assignment given: 20. January 2007
Supervisor: Svein Erik Bratsberg, IDI

Abstract

This paper describes the work done to add hot standby replication functionality to the Apache Derby Database Management System.

The Apache Derby project is a relational database implemented entirely in Java. Its key advantages are that it has a small footprint and it is based on the standard Java JDBC and SQL standards. It is also easy to install, deploy and use as well as it can be embedded in almost any light-weight Java application.

By implementing a hot standby scheme in Apache Derby several features are added. The contents of the database is replicated at run time to another site providing online runtime backup. As the hot standby takes over on faults availability is added in that a client can connect to the hot standby after a crash. Thus the crash is masked from the clients. In addition to this, online upgrades of software and hardware can be done by taking down one database at the time. Then when the upgrade is completed the upgraded server is synchronized and back online with no downtime.

A fully functional prototype of the Apache Derby hot standby scheme has been created in this project using logical logs, fail-fast takeovers and logical catchups after an internal up-to-crash recovery and reconnection. This project builds on the ideas that are presented in Derby: Write to Neighbor Mode[4].

Preface

The work in this project is done as a Master Thesis in the 10th semester of the Integrated Master's Degree (Sivilingeniør) with the Department of Computer and Information Science at NTNU 2007. I would like to extend thanks to Professor Svein Erik Bratsberg for his great advice and invaluable help during the project. I would also like to thank Jørgen Løland and Øystein Grøvlen at Sun Microsystems, for their help both to implement the needed changes to the existing Derby system and valuable feedback on this report.

I expect this project to result in a fully functional prototype of a hot standby replication scheme for Apache Derby. The prototype will most likely suffer a small performance drop due to the added complexity, but with some care and synchronization this penalty should not be too severe.

This project is an extension of Derby: Log to Neighbor Mode[4]. The chapters 3 and 4 have some minor changes from that project while the rest of the report has major changes.

Trondheim, June 3, 2007

Egil Sørensen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Derby Overview

Apache Derby is an open source Database Management System (DBMS) written in pure java, which means that the system is compiled once and can run anywhere with the help of a Java Virtual Machine (JVM). It also has a small footprint, the jar file is only 2 MB, meaning that it has significant useful functionality in a small code size with efficient runtime resource usage.

Derby was started as a java database project named JBMS by Cloudscape Inc, a newly founded company with main focus on developing Java database technology in 1996. JBMS was renamed to Cloudscape and released on the market in 1997 with new versions released roughly every six months. In 1999 Cloudscape was acquired by Informix Software, Inc which was in turn acquired by IBM in 2001. The product was then re-branded IBM Cloudscape and the following releases served as embedded databases used in IBM's Java products and middleware. In August 2004 IBM contributed the code to the Apache Software Foundation as Derby and in July 2005 the product graduated from the Apache Incubator into the Apache DB project. Sun Microsystems, Inc adopted the Derby project and released a distribution of their own, the Java DB.

Today, the Apache Derby is available as Apache Derby, IBM Cloudscape and Java DB and it shows great promise especially for the possibilities added with the embedded environment it supports. Since the footprint of the jar file is so small it can be embedded into almost anything and it is therefore commonly used within web applications, development environments and portable devices.[16]

1.2 Specification of project

After a few meetings with Svein Erik Bratsberg we came up with the following specification of what should be done in this project:

1. Ship the logical log to another Derby node in hot standby mode
2. Save the logical log to disk on both nodes
3. Redo all logical log records received by the hot standby
4. Perform takeover if the primary database should fail. Both the primary and hot standby should have fail-fast semantics.
5. The primary must provide a catchup service to the hot standby when it reconnects after a failure.
6. Measure the results in means of throughput, response times and hardware utilization and compare these to the original Derby network server database.

Chapter 2

Transactions and Logging

In this section we will take a closer look at some of the fundamentals regarding replication and logging. The ACID properties of a storage system or database is often taken for granted. However, all operations done to a database or other storage system does not necessarily possess all of these properties. To make the ACID properties a general commodity the transactions were introduced.

2.1 The ACID Properties

The ACID Properties are the cornerstones of database management systems (DBMS). If they are not upheld the DBMS risk losing committed information and corrupting data leaving it in an inconsistent state. The four properties are: [1, 3]

- **Atomicity** A transaction is seen as atomic it if goes from the initiate state to the final state while showing all of the work being done in between as a single, atomic operation or if the operation was never done at all. This "everything or nothing-at-all" rule ensures that no incomplete or inconsistent data can ever be returned to the user.
- **Consistency** The consistency property ensures that a transaction goes from one consistent state to another. If the application running deems the new state inconsistent, the transaction is aborted.
- **Isolation** A transaction needs to be isolated from other actions to protect it from dirty data. Various locking mechanisms are efficient tools to make the transaction feel that it is running alone on the required data. No dirty, uncommitted data written from other transactions must be read by the running transaction to ensure isolation. The Isolation property is very closely related to Atomicity in that it hides all intermediate results.
- **Durability** When a transaction is committed, its changes to the database are made durable, in such that they are not lost.

2.2 Transactions

A transaction is a collection of actions that together have the ACID properties covered[1]. All of the actions within a transaction do not need to have covered all the ACID properties as long as the transaction can compensate for this in some way. In this section we will take a closer look at different types of operations and how these are combined by a transaction to ensure the ACID properties are upheld. A transaction can have two possible outcomes, either commit or abort. If a transaction commits, all the changes are saved and made available to the rest of the database system. If a transaction aborts however, all of its changes are undone and its locks are released. In this way a transaction is both atomic, isolated and durable. If the transaction is consistent or not is however the responsibility of the application using the database.

We can divide all the possible actions into three groups: Unprotected actions, Protected Actions and Real Actions[3]. Unprotected actions are actions that lack all of the ACID properties except consistency. These actions can fail at any time and if transactions are to use unprotected actions, they must be protected by the transactions with compensating actions if it fails and needs to be undone or redone. An example of an unprotected action is a single disk write. When a block is written to a disk there is no guarantee for the outcome. Logged writes are a usual approach to compensate for this unprotected action and it is described later. Protected actions are actions that are reliable and have all the ACID properties making these actions are safe building blocks for normal transactions. Real actions are actions that affect the system in such a way that it is very hard or impossible to undo its changes. If real actions are to be used in a transaction it is important that the transaction can guarantee that it will commit before the real action is performed. In worst case the real action must be undone by a compensating action, but as stated earlier this is either very difficult or impossible. How real actions can be included in a transaction can be seen in Figure 2.1 where *DRILL_HOLE* is the real action being deferred until the transaction is guaranteed to commit.

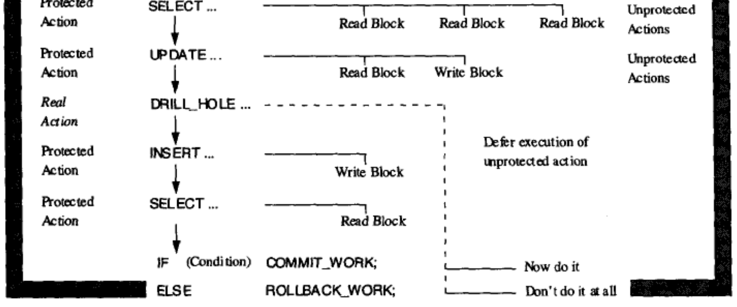


Figure 2.1: Protected versus real actions in a transaction-oriented evironment [3]

2.3 Logging

To ensure that disk writes have the ACID properties it is important that disk write actions fall under the protected actions category mentioned in the previous section. To be sure that the disk write is both atomic and durable it is necessary to employ a writing mechanism that can guarantee the follow two criteria:

1. The disk write is either done successfully or not done at all
2. The disk write is durable. This means it cannot be reversed due to system failures after it is written

2.3.1 Physical Logging

One approach to ensure the above criteria is the *logged write* approach. For every write to a disk page the old and new values for the page is saved on a log external to the original data. In this way the write can be both undone and redone if the write is not successful or the action is forced to abort. The log can also be used for data replication as well as all the page values are reflected in the log as well as on the disk. This log can then be used to rebuild the database after a database failure. To log single blocks like this is known as Block Oriented Logging [2]. The main benefits from the block oriented logging is that not only data are logged, but also index pages and the B-Tree structure of the index is logged. This means that it is easy to revert changes done both to the data itself and the indexes to an earlier state. Another great asset to the physical logging is that both the undo and redo operations are always idempotent. This is because these operations always put the page back to the previous state.[3]

While block oriented logging can guarantee the needed ACID properties it still has some drawbacks. For each change that is done to a page on a disk the entire before and after image of the page must be stored in the log. The log itself also has references to the affected disk pages and this makes the log hard to distribute for replication purposes.

2.3.2 Logical Logging

The physical log described earlier has references to the affected disk pages. If the log is to be distributed to other database systems, these need to have identical physical disk layout. This is in many cases an unreasonable assumption to make. A logical log however only saves the logical operation along with its redo and undo mechanisms instead of the changes made to every single block affected by the operation.[3] This is a highly desirable property especially for replication purposes. If other database systems have got the same database schema[1], the logical log can be used to distribute and replicate the data directly. Another interesting consequence derived from distributing the logical log is that the neighboring database does not even need to be run on the same operating system or be the same type of DBMS as the one distributing the log as long as it has the same

[1]The same tables and fields

schema as the log distributor. If the neighboring database has the same operating system, database system and disk layout however, the physical log can also be used for replication purposes. As mentioned earlier the index pages and B-Tree structures are logged as well. If the physical log is used for replication purposes the two databases will be identical. With logical logging however, they are equivalent in that they have the same values for all of its tuples. The index structure will most likely be different though.

Another very desirable property of the logical log is the drastically reduced log size. In a naive implementation of block oriented logging the log will be at least twice the size of the database record it was created for. Let us take a look on an example: If a block B_b is 4 kB large and the data written is about 300B this means that the log record is at least $2B_b = 8kB$. This means that the log is roughly 13 times as big as it could have been if only the written data was logged. In the worst case where the record is distributed with small parts over many blocks on the disk the amount of data written to the log is increased with $2B_BN_B$ where N_B is the number of blocks the record is split over. One of the solutions to this problem is to employ Record Oriented Logging[2]. With record oriented logging the record itself is being logged with a before and after image instead of logging all of the disk blocks this record resides on. In this way the maximum amount of data written to the log for a record R is 2R and a small overhead of the log record itself.

However, this solution still suffers from recording the whole before and after image of the entire record. If only one of one hundred fields of a record is changed it should not be necessary to log the entire record. To reduce the log size to a minimum only the logical operation along with logical undo/redo statements should be logged, this is called *logical logging*. The reason for this is that the log consists of logical operations (e.g. update record <R> set field <F> = value <V>) instead of physical operations (write data <D> to disk block <B>, page <P>). The logical log is not concerned by the physical layout of the disk and this also a strength for replication purposes. In this way the log can be used to replicate the data on a completely different DBMS as long as the table structures are the same.

As we have seen, logical logging is a cheap method[2] to ensure the ACID properties and also provide the system with good replication possibilities to other systems regardless of the disk layout, hardware or operating systems. However, the Logical Log has two important drawbacks, namely *partial actions* and *action cosistency*[3]:

- **Partial Actions** A logical insert may invoke triggers or other mechanisms while running. If the database crashes while this insert is being done, the action is left in an inconsistent state, or half done. When the undo is attempted at database restart it is not presented with an action-consistent

[2]Regarding disk usage, both in size and to minimize utilization of the disk for logging purposes

state.

- **Action Consistency** Simple actions with simple writes or messages are atomic, in that they are either done completely or not done at all. However with complex actions involving multiple messages or writes are not necessarily atomic. If a partial complex action is presented to a undo or redo at database restart this is known as an action inconsistency.

In this project the logical log is chosen for replication purposes in addition to the physical node-internal log in both Derby databases. The logical log will consist of simple actions, avoiding partial actions and action inconsistency and physical log will be used for normal internal recovery. For more details on the log design choice see chapter 5.

2.3.3 Write-Ahead Logging

A refinement of the logged write described earlier is the Write-Ahead Logging (WAL). The WAL protocol ensures two very important properties regarding crash recovery:[7]

1. All log records containing changes to a certain page must be written to a stable storage before the page itself is written to disk.
2. All log records belonging to a transaction must be written to a stable storage before the transaction is allowed to commit

These two properties combined ensure the Atomicity and Durability ACID properties. If a transaction is recorded to be committed, WAL insures that all the log records belonging to this transaction is stored safely as well. In this way the transaction can be redone in the events of a crash and can therefore be said to be durable. With WAL all updates to the database can be done in-place.

2.4 ARIES

Algorithms for Recovery and Isolation Exploiting Semantics (ARIES) is a refinement of the WAL protocol. It has a STEAL/NO-FORCE buffer policy meaning that pages can be overwritten even if the owner transaction has not committed (dirty data is allowed in the database) as well as updated pages are not forced to disk upon commit. In ARIES every log record contains a unique log sequence number (LSN), a reference to the owner transaction and a reference to the previous log record for this transaction. Each data page also contain a recoveryLsn which points to the log record of the last update done to this page.

During normal operation ARIES maintains two control structures: [8]

- A transaction table containing the identification of each running transaction and it's status.

- A dirty page table containing a reference to every dirty page in the system

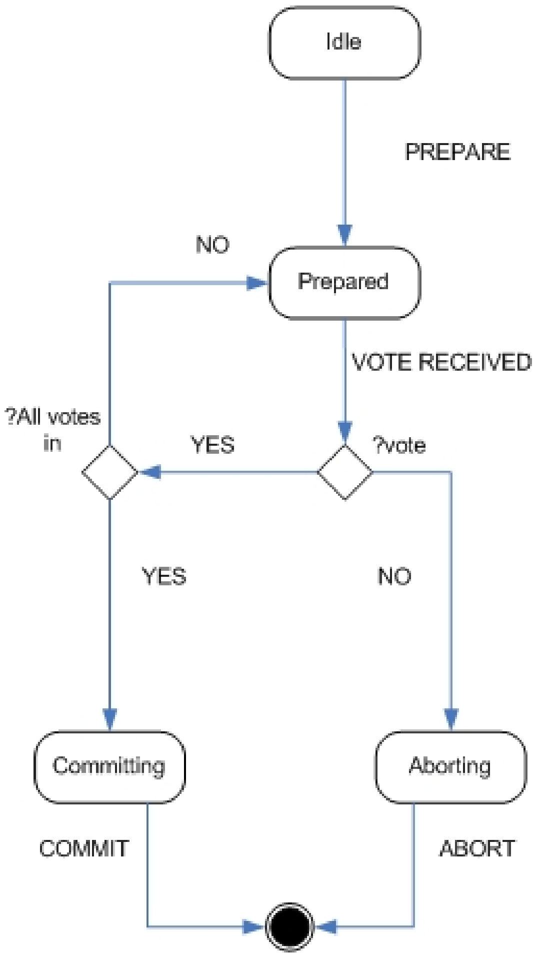
Periodically ARIES creates checkpoint entries in the log containing the transaction table and the dirty page table to ensure that they survive a crash along with the rest of the log

After a crash, ARIES has a 3-step recovery algorithm which consists of the Analysis, Redo and Undo phases. During the analysis the log is checked for the latest checkpoint and the dirty page table and transaction table are fetched and rebuilt. The log is the scanned from the checkpoint and the transaction table and dirty page table are both updated accordingly. If a transaction commit or abort log record is found in the log the transaction is removed from the transaction table. If a transaction begin log record is found it is added to the transaction table. When the state of the database is restored the first recoveryLsn from the dirty page table is chosen as the firstLsn for the recovery. The next step is the redo phase where the log is scanned from the firstElement and every log record is redone if needed. After the redo is completed an undo phase begins where all operations from non-committed transactions are being undone. For each undo-operation that is done a Compensating Log Record (CLR) is created. The CLR is a dummy record to ensure that the undo-operation is only done once, it's previousLsn is set to the same as the previousLsn from the log record it is compensating for. If another crash occurs the CLR will ensure that the recovery process skips the log record it is compensating for. After the undo-run is completed ARIES writes a new checkpoint and returns to normal processing again. For more information about ARIES and it's recovery scheme see [7, 8].

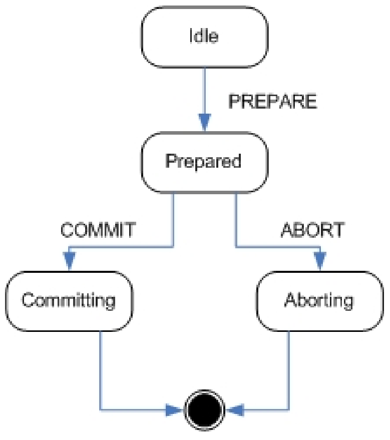
2.5 Two-Phase Commit Protocol

Commits are operations that are required to be atomic and durable. How to enforce these two properties on a centralized DBMS is described above by using ARIES. However, in distributed environments where the transaction can be running on multiple network nodes simultaneously this task is not trivial. To ensure the ACID properties it is required that either all or none of the distributed nodes commit the transaction.[9] In the distributed environment one of the processes is connected to the user application, called the coordinator. The other processes are hereby referred to as the subordinates.

When the coordinator decides to commit the transaction a PREPARE message is sent to all the subordinates telling them to prepare to commit the transaction. Every subordinate then returns a vote if the transaction should be committed or not and await the final reply from the coordinator. If any of the subordinates transmit a NO-vote this acts as a veto and the transaction cannot be committed. The coordinator then goes into the aborting state and tells all of its subordinates to abort the transaction as well. However, if all the subordinates vote YES the coordinator enters the committing state and sends a COMMIT-message to all of its subordinates. When all the commits have been acknowledged a force-commit



(a) Coordinator Node



(b) Subordinate Node

Figure 2.2: State Diagram for Normal 2PC Behaviour

is written at the coordinator and the transaction is committed on all of the nodes. The normal message flow between the coordinator and one of its subordinates and their states can be seen in Figure 2.2 and Figure 2.3.

Some modifications have been made to optimize the Two-Phase Commit (2PC)-protocol including a Presumed-Abort and Presumed-Commit 2PC protocol. Both are thoroughly described in [9] . In this project a very simplified 2PC Presumed-Abort will be used in the two-safe replication approach.

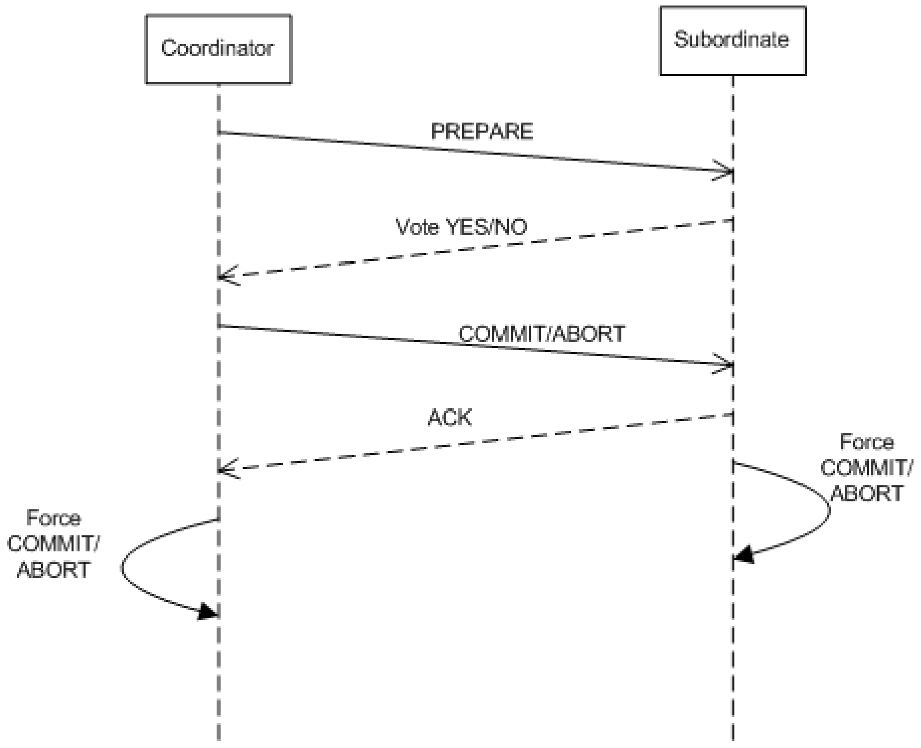


Figure 2.3: Normal Message Flow in 2PC

Chapter 3

Replication and Log Distribution

One of the primay goals in this project is as mentioned earlier to ship log records to a neighboring hot standby database. This chapter will take a look at some of the potential uses for this distributed log. When it comes to replication, transactions can have two different properties, either *one-safe* or *two-safe*[6]. The transaction is said to be two-safe if all the log records concerning this transaction are forced to the neighbor node before the transaction is allowed to commit. The opposite, one-safe sending, is when the log is saved and sent over in batches either at a given time interval or when the log buffer fills up. In this way the log can be sent at times where the server and/or network utilization is low. To keep the following examples simple we assume that the system consists of one primary and one secondary database.

3.1 Data Replication

The simplest use of the distributed log is for data replication purposes. The log can either be stored in its original form as a backup or it may be handled by another DBMS. In the latter case the DBMS receives the log records and processes them accordingly. When a commit record is received for a transaction x, the whole transaction is done at the backup DBMS and committed. If the primary database is completely lost, the backup database can be used as the new primary when a new system is built. There are two ways to ship the logical log, as mentioned earlier: By using one-safe and two-safe replication. Both have a different set of properties that makes one more desirable than the other in a chosen environment.

3.1.1 One-Safe Replication

To achieve higher throughput from the system and to reduce the response times, the running transactions are allowed to commit their changes to the primary database as usual without synchronizing with the secondary database first[6]. This is also called *a-synchronized* replication or *one-safe replication*. For the transaction controller the replication is completely transparent and the users of the database system are not affected by transmit delays to the secondary database. However, the one-safe replication approach has one major drawback:

The primary database could crash before or while transmitting the log to the secondary database risking loosing transactions entirely. The potential problem can be seen in Figure 3.1:

1. The transactions are running normally, T1 is committed and the changes are propagated to the secondary database
2. Transaction T2 commits on the primary
3. The primary goes down before the commit record for T2 arrives at the secondary.
4. The secondary becomes the primary without redoing T2 as T2 was not recorded committed
5. The changes done by transaction T2 is lost, durability is compromised

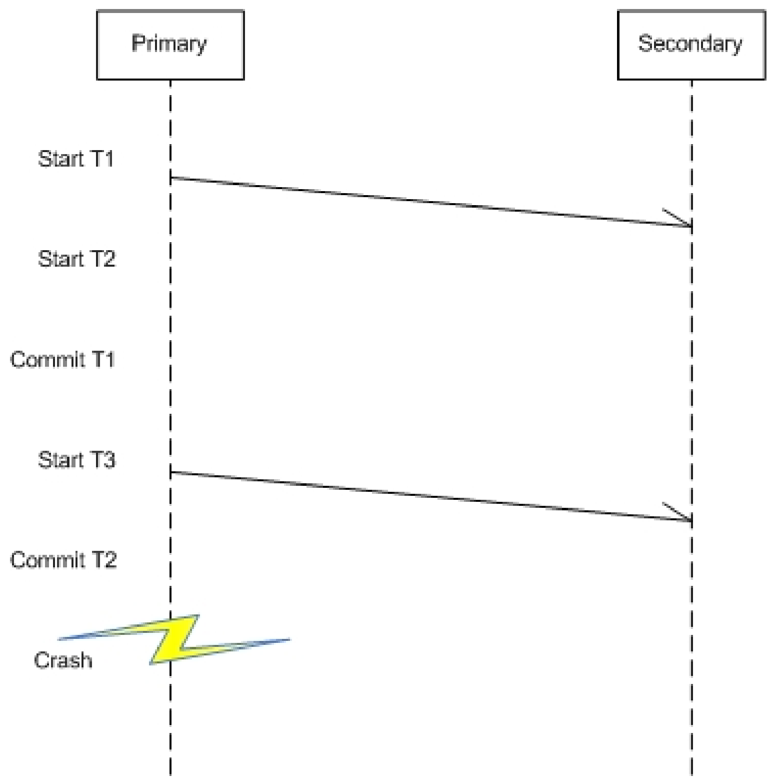


Figure 3.1: Potential problem with one-safe replication

As shown above, loss of transactions in one-safe replication can pose a serious problem and can in worst case compromise the durability property.

3.1.2 Two-Safe Replication

An alternative to the one-safe replication method is to make all transactions wait until the changes are propagated onto the secondary database before they can commit. When the log has been confirmed received by the secondary the transaction is committed using a 2PC protocol. While the two-safe approach guarantee the ACID properties it also imposes a performance problem: The response time for transactions are increased by at least one round time of messages.[6] Since the 2PC protocol does not release its locks until it is committed it does not only increase the response time for transactions, but it also increases lock contention at the primary which in turn may dramatically affect the performance of the entire system.[5]

To illustrate this, take a look at two-safe replication during normal execution in Figure 3.2. There are two transactions active that depend on each-other[1]. As we can clearly see the response time for both T1 and T2 will be dramatically increased from the one-safe replication example shown earlier as T1 needs to wait for confirmation from the secondary before performing a 2PC and T2 needs to wait until T1 is committed until it can start at all. With large hot spots like these in databases this approach will impose serious performance penalties. However, when the primary crashes after the prepared commit of T2, T2 is not entirely committed when the crash occurs. In this way T2 will be aborted by the secondary upon no doCommit confirmation from the 2PC and it will eventually be aborted by the primary when it goes online and starts the recovery process.

Another problem with the two-safe replication approach is when the secondary database crashes or the communication lines goes down between the primary and secondary databases. In a naive implementation of the two-safe approach transactions will still wait for the secondary to respond before they are allowed to commit and release their locks. Arriving transactions would pile up until either the secondary would come back online or the primary cannot receive any more transactions. At best the throughput and response times would be awful. In worst case the primary would appear as offline after a while, rejecting connections. However, the ACID properties are still enforced by both the primary and secondary databases.

An optimized two-safe replication approach has also been researched to achieve the desirable properties of the two-safe protocol described here optimized by addition of partial synchrony in the log transfer protocol[5, 6]

[1]They need exclusive locks on at least one of the same objects in the database

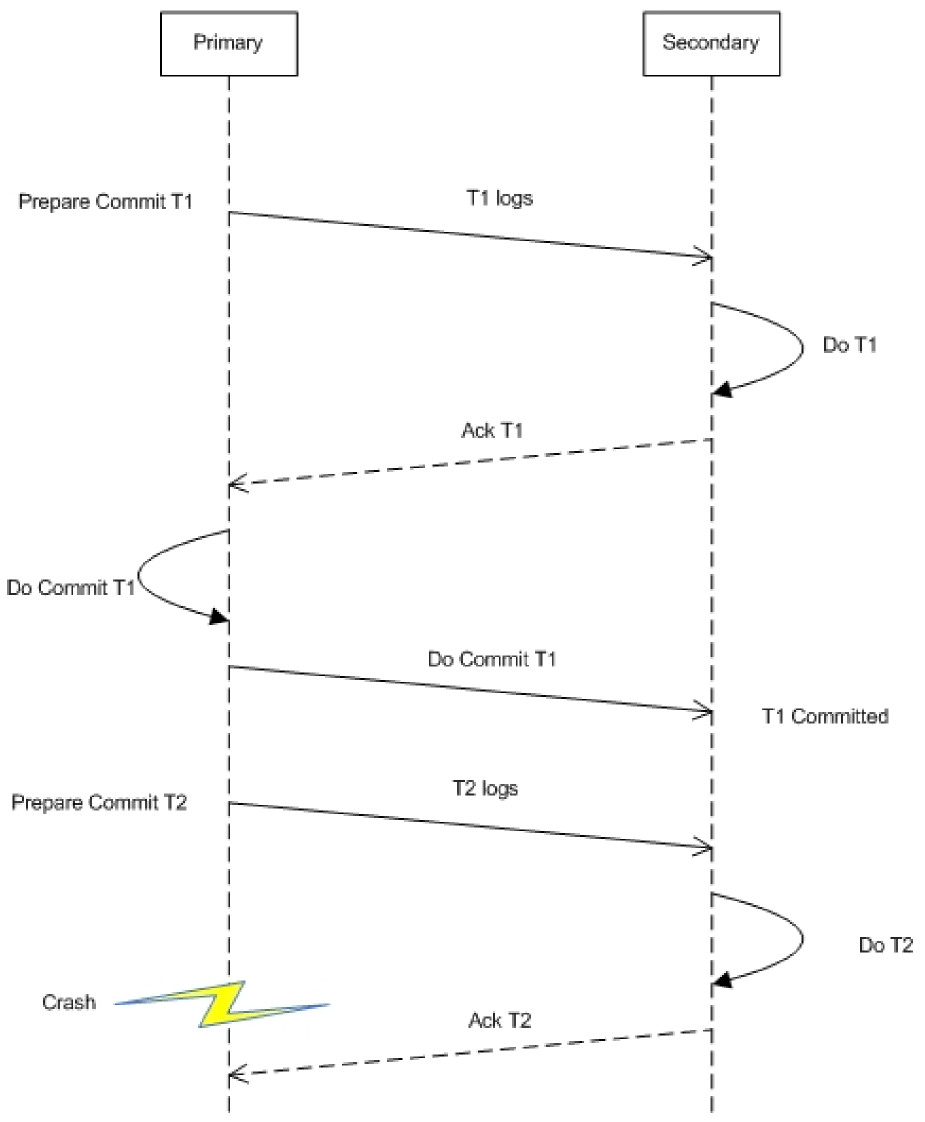


Figure 3.2: Two-safe replication during normal execution

3.2 Hot Standby

In systems that require high availability it is important to remove any single point of failures. If a process goes down it might take the whole system down as well, or at least the processes that depend on it. To avoid the single point of failure it is important to have redundancy in processes, this goes for database systems as well.

In the hot standby case the DBMS consists of two identical databases: One active primary and one passive in hot standby mode[3]. This means that the active database answers all the requests from the clients and processes these like usual. However, should the primary fail, the database in hot standby takes over as the primary while the old primary is set to restart and do recovery work. When the crashed database comes back online it is either set in hot standby-mode while the new primary answer all the clients or it tells the now active database that it is ready to go and takes over as the primary again while the other goes back to hot standby.

To make the hot standby ready to take over for the old primary it is important that it is synchronized with the primary and this is where the distributed log is needed. For every operation done on the primary, the log record of this operation is sent to the hot standby and redone there as well. However, when transmitting only the log records to the hot standby, the transactions running when the crash occurs will loose their work as the hot standby does not have the transaction table or lock table. This means that all running transactions must be aborted upon a primary crash, unless a monitoring process can restart the transactions on the hot standby node. It is also important that the primary is fail-fast such that the hot standby can take over after a small delay and that the primary crash is masked as a slight delay only to the connected client transactions.

3.3 ClustRa

In parallel database systems where high throughput and short response times are very important it is not feasible to force the log to disk on every commit. ClustRa is a highly available parallel database which employs a main-memory version logging technique, *neighbor Write-Ahead Logging* (nWAL) combined with a node-internal log.[2, 12] The ClustRa database is divided in two databases running in parallel with a number of nodes with independent failure nodes on each site. On each node there is a transaction controller, a database kernel, a node supervisor and an update channel. ClustRa employs two-safe replication with an early answer to counter the extra time it takes to commit using 2PC: On normal execution of a request the transaction controller appoints a hot standby transaction controller on a node on the other site along with a database kernel on the nodes containing the main and hot standby-replica of the data being changed. The transaction controller then sends the operation that should be done to the kernels on both the primary and hot standby node along with the number of log records the hot standby-node should receive. The update channel on the primary

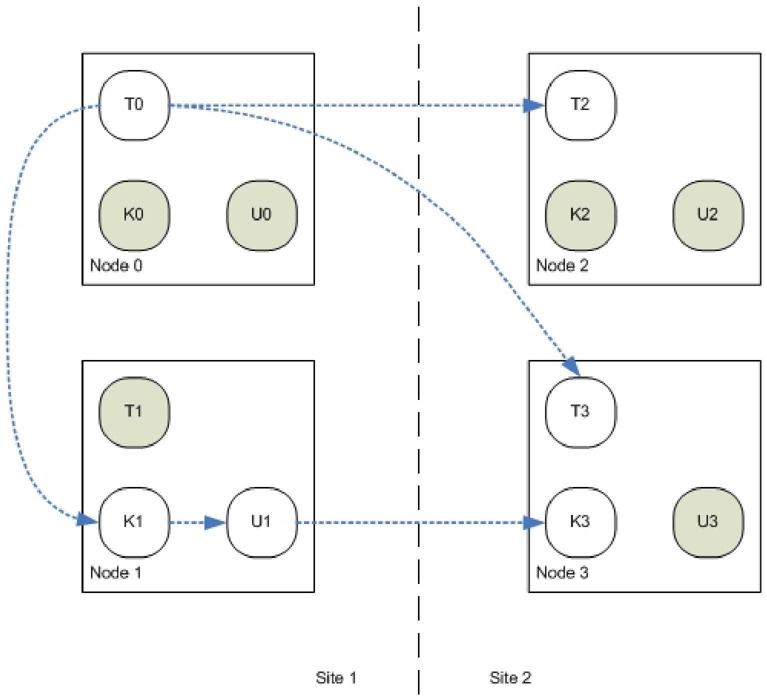


Figure 3.3: Two-safe replication with nWAL in ClustRa

node sends the log records over to the hot standby-node where they are redone. After both the kernels have sent an ack to the active transaction controller it submits an *early answer* back to the client and then initiates the rest of the commit procedures. The above example is illustrated in Figure 3.3, where T0 is the active transaction controller, K1 and U1 is the active database kernel and update channel while T3 is he hot-standby transaction controller and K3 is the hot-standby database kernel. As shown, the distributed log must be written to two nodes with independent failure nodes before the changes are made to disk and before the transaction is committed. By logging to both sites an entire site can crash and the changes will still be reflected on the system.

Chapter 4

Derby Design

The architectural design described on the official Apache Derby website[15] is divided into two different views, both will be presented in the following chapter.

4.1 Derby Layers

The Derby system can be divided into layers of functionality to give a overview of how the different components work together and also to group similar functionality together. The layers in the bottom provide services for the layers above while concealing the underlying data structures and data. As shown in Figure 4.1 the layer/box model consists of four layers: *JDBC*, *SQL*, *STORE* and *SERVICES*.

4.1.1 The JDBC Layer

The Java Database Connectivity (JDBC) layer is the top layer of Derby and is also the only layer that provides an Application Programming Interface (API) to other applications or users. This single point of entry ensures that applications using the Derby database is forced to access it through the JDBC layer. They can only use prepared statements through the standard java.sql.PreparedStatement class and not through any of the internal Derby classes.

4.1.2 The SQL Layer

The Structured Query Language (SQL) layer is responsible for the compilation and execution of SQL queries. The compilation process can be described by five steps:

1. Parse the query into a tree of query nodes using Javacc
2. Binding the tree nodes to the dictionary to get table names, column names and so on
3. Optimize the access path selection through the tree
4. Generate a statement plan. This plan is directly compiled to byte code.
5. Loading the generated class and create an instance of this class to represent the state of the query

When the compilation is done the statement plan is executed on the object created in the compilation process. This execution returns a native Derby resultset that is passed back to the JDBC layer. The JDBC layer then converts the resultset into a standard JDBC one and represents the results to the client application. The resultset objects sent to the JDBC layer interface with the Store layer to fetch data from the raw data store.

4.1.3 The Store Layer

The store layer is responsible for handling the data storage and is divided into the access and raw sub-layers. The access layer provides the SQL layer with a row based conglomerate interface. In this way the SQL layer can view the underlying raw data store as a traditional database store with rows and columns. The raw store is responsible for storage of the data on disk pages as well as transaction logging and transaction management.

4.1.4 The Services Layer

The services component is a collection of utility modules used in the entire system. Some services examples are error logging, cache management and lock management. Because of the lack of clear interfaces between the services layer and the others it is put aside the other layers instead of inside the hierarchy as usual. Client applications may however not connect to the services layer, the only external interface is as described above through the top of the JDBC layer.

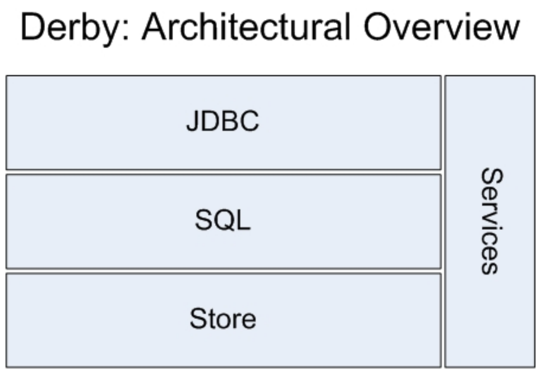


Figure 4.1: Derby: Architectural Overview

4.2 The Module View

Another approach to the derby architecture is to divide the source code into modules. A running system can be described as comprised by a monitor and a collection of modules.

4.2.1 The Monitor

A monitor is a way to map external module requests to internal implementations. In this way the monitor can map different internal versions to different external requests with regards to the request itself and the environment of the request. An example is when a client connects to the JDBC layer as described above, the Derby system recognizes the version of the Java Virtual Machine (JVM) of the client. This is then used to determine what JDBC version should be used.

4.2.2 Modules

A module is a set of functionality and is defined by the modules interfaces. The module is often seen as a black box to the outside world, what happens inside the module is irrelevant as long as the module responds correctly to incoming requests. The module approach ensures that it is easy to replace modules or write new ones as long as the interfaces are well designed. It can be safely replaced without affecting the callers code.

The easiest way to see the source code in Derby as modules is to divide the source code according to the internal packages in Derby:

Module	**Derby Package**
Network Server	org.apache.derby.drda **and** org.apache.derby.impl.drda
JDBC	org.apache.derby.jdbc **and** org.apache.derby.impl.jdbc
Execution	org.apache.derby.impl.sql
Access	org.apache.derby.impl.store.access
Lock Manager	org.apache.derby.impl.services.locks
Buffer Manager	org.apache.derby.impl.services.locks
Logging	org.apache.derby.impl.store.raw.log
Data Store	org.apache.derby.impl.store.raw.data
Transaction Control	org.apache.derby.impl.store.raw.xact

Table 4.1: Derby Modules

As we can see from Table 4.1 there is a connection between the module view and the layered box view. The logical log implemented by this project will be added as another module in the store layer and will have interfaces toward the execution part of the SQL layer where the logging is performed.

Chapter 5

Design Choices

In this chapter we will take a look at the most important design choices

5.1 Design Goals

To implement a Hot Standby Scheme in the Derby DBMS there are some changes that need to be done. The following functionality is needed:

- **Logical Log**
  A Logical Log must be created to make the database log not dependant on the physical layout of the disk. Also when the log is logical only the appropriate before and after images of the operation need to be written to the log, not the entire affected disk pages. The logical log is to be used for replication only and will not be used for any recovery purposes other than catchup. The physical log of Derby is used for node-internal transactions and recovery as usual, and thus the problems with partial actions and actions consistency are avoided as the logical log is only used to log database tuples.
- **Logical Log Shipper and Receiver**
  A mechanism to transfer the logical log between the primary and the hot standby database must be created.
- **Fail-Fast Mechanism**
  To avoid long down times it is important that the Hot Standby have a way to detect that the primary database has died. To implement a fail-fast mechanism a heart-beat signal is needed between the two nodes. If several heart-beats are missed the primary is assumed to be down.
- **Takeover Mechanism**
  When a primary database dies it is important that the hot standby is able to take over. The takeover can be summarized in the following four steps.

  1. Discover primary down
  2. Abort running transactions
  3. Switch internal mode to primary

4. Accept connections

- **Hot Standby Catchup**
  When the former primary database comes back online after the crash it needs to be caught up with the new primary. When the hot standby database is down for some reason the primary database stores its logs on stable storage while waiting for the hot standby to become active again. When it does the logical log saved on stable storage must be sent to the hot standby until it is up-to-date and normal processing can continue. Before catchup the hot standby is made up-to-crash consistent before it becomes available. The node-internal physical log is used for this recovery as usual in derby. When the hot standby is finally recovered the catchup can commense.

5.2 Design Overview

To avoid too many changes inside the Derby core, most of the functionality is created as services that are started whenever needed. The client connections to the database goes through the network connection control in Derby and are passed on to the embedded database. The network interfaces used by the hot standby implementation should be placed within the network connection control. However, due to the design of the Derby system, the communication between the network service and the embedded database is one-way only (except for the jdbc-connection), so the log shipping and receiving still have to be included in the derby core. The overall design of the system, including one connected client, a primary database and a hot standby database can be seen in Figure 5.1. The client connects to the network server as usual so there is no need to make any changes in the client software. The server of the primary database then start the connection and connects to the hot standby server. The network servers exhange heart-beats and forward the statements from the client to the Derby core. The core is responsible for creating the logical log and shipping it to the hot standby where it is redone. The network server is also responsible for setting states in the core, these states include if the hot standby is alive or if the server is in hot standby or primary mode.

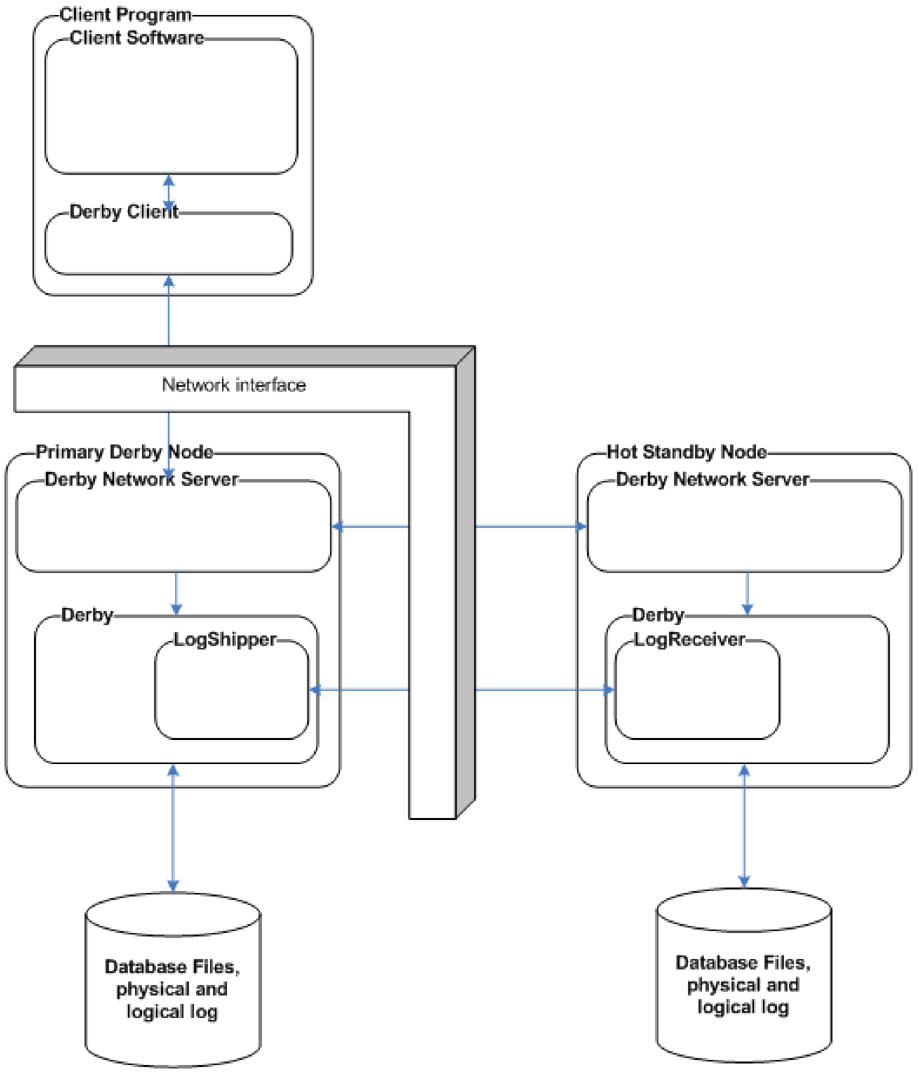


Figure 5.1: Design Overview

5.3 Logical Log Format

The logical log record is based on the *DO-UNDO-REDO* protocol described in [3] and should in our case contain the following fields:

- Log Sequence Number (LSN): The LSN is a unique field containing the sequence number for this particular log record. LSN's are needed to ensure the ACID properties when using the ARIES approach to recovery.
- Previous LSN for this Transaction (prevTransLSN): The previous LSN for this transaction is needed by the recovery agent to roll back the changes done by a particular transaction.
- Transaction id: The identification for the particular transaction doing this operation.
- Operation: The type of the operation done is saved in the operations field. This is necessary both for undo and redo information.
- Table and Record id: The table and record id's currently being changed by the logged operation. The table id can be either the name of the table or some other unique id. The record id is the primary key of the record being changed.
- State information: The state information is the before and/or after images of the affected records. In our case the log records can be either Undo-only, Redo-only or Undo-Redo log records. This means that the log records stored are smaller and log size is reduced. On insert operations only the after image is needed, while delete operations only need the before image. Update operations however require both.

One of the main reasons to employ logical logging in this case is to make replication to other DBMS nodes possible. As a consequence we can also manage to reduce the log size as explained in subsection 2.3.2. To keep the log as small as possible only the changed values are included in the log records. This means that if we do a query:

UPDATE personInfo set city = 'New York'

We only have to store the key to every post along with the old and new city instead of storing every single personInfo-record in the log. Considering commit and abort operations, these log records contain only the LSN, TransactionId and Operation fields described above.

The complete overview of the new log format can be seen in Table 5.1

LSN		
prevTransLSN		
TransactionId		
Operation		
TableID		
Record		
States	Insert	After Image Only
	Update	Before and After Image
	Delete	Before Image Only
	Commit	No Images
	Abort	No Images

Table 5.1: Logical Log Format

5.4 Logical Log Implementation

To implement a new logical log in the Derby system a new logical log module is created. The main contents of the module are a Logical Logger service, the Logical Log object along with their interfaces. The Log Shipper and Receiver are implemented as a new 'net' service in the service layer.

5.4.1 Logical Logger

The logical logger will be a simple service containing the log method. The rest of the functionality regarding logging and shipping will be handled inside the Logical Logger to keep the implementation clean and to change as little as possible of the existing Derby system. The two-safe support added to the system is very simplistic and only ensures that the log records arrive on the backup before the transactions are allowed to commit.

5.4.2 Logical Log

The logical log is created as a linked list of log records. For simplicity the log is used and shipped as an object in the java heap instead of translated to byte code in the way the physical log in Derby is written.

5.5 The Hot Standby Controller

To keep track of the hot standby states a hot standby controller is needed. The hot standby controller is also responsible for internal database connections used by redo of arrived logical logs, and to ensure database consistency during takeover and catchup. The hot standby controller is also the interface the network server uses to set the states and initialize takeover as shown later in this chapter.

5.6 Replication Modes

There are two different replication modes to take into concideration in this project. The one-safe and two-safe replication schemes as mentioned earlier. The one-safe replication mode is rather straightforward. The primary ships the logical logs at chosen intervals and is redone at the hot standby node. The two-safe approach however requires that a 2PC protocol is used, where the primary is not allowed to commit until the hot standby has signalled to do so. In this project the main focus is set on one-safe replication. However a two-safe-like replication scheme is added where 2PC-messages are handed between the primary and the hot-standby before the log is shipped, to simulate two-safe replication. The 2PC scheme used is presumed-abort[9]. This means that an abort does not need to use 2PC because all transactions that are not committed are sooner or later aborted. If no commit is received at the hot standby the transaction is aborted.

5.7 Communication

There are multiple alternatives concerning which protocol to use for communication:

- TCP
- UDP
- Java RMI
- SOAP

In our case the most reasonable approach would be to use the TCP-IP protocol. If we are to use UDP it is more difficult to make guarantees of packages arriving in the right order or arriving at all. Java RMI uses the RPC approach, that is a request is sent and an ack is received for every call. This provides a lot of unnecessary overhead and will most likely act as a bottleneck in our communication. In addition to this Java RMI must have a dedicated broker to register services, which is not necessary when the receiver node does not move from one network address to another very often. SOAP suffers from the same problems, but in addition the SOAP messages have a very large xml overhead which renders the protocol way too slow and cumbersome for our use.

To ensure that the network overhead is as small as possible a TCP-connection is set up on startup and kept up for the duration of the database server. If the connection goes down reconnect attempts are made until the connection is up again.

5.8 Communication Interfaces

To simplify the connection between the two derby nodes a message based request-response protocol is needed. The primary must contain a client which sends messages containing logical logs, heart-beat messages and other requests while the hot standby node must contain a server which answers these requests.

5.9 Network Server

To implement the hot standby scheme some changes must be made to the Derby Network Server as it is responsible for the client connections.

5.9.1 Initial Startup

When the database is initially started one database is started in hot standby mode and another is started as a primary. The primary connects to the hot standby database at the network address provided with a parameter when the database is started from command line. When the hot standby gets a connection from the primary the network address of the primary is saved so that the hot standby knows where to find the other database if a takeover is necessary.

After this the primary goes on and accepts connections from clients. The hot standby does however not accept connections until it becomes primary and takeover is finished. The possible states and their transitions can be seen in Figure 5.2. As shown the transitions are start → hot standby → primary → end. There is no way for a hot standby to do a take-back, that is go back to primary after becoming a hot standby.

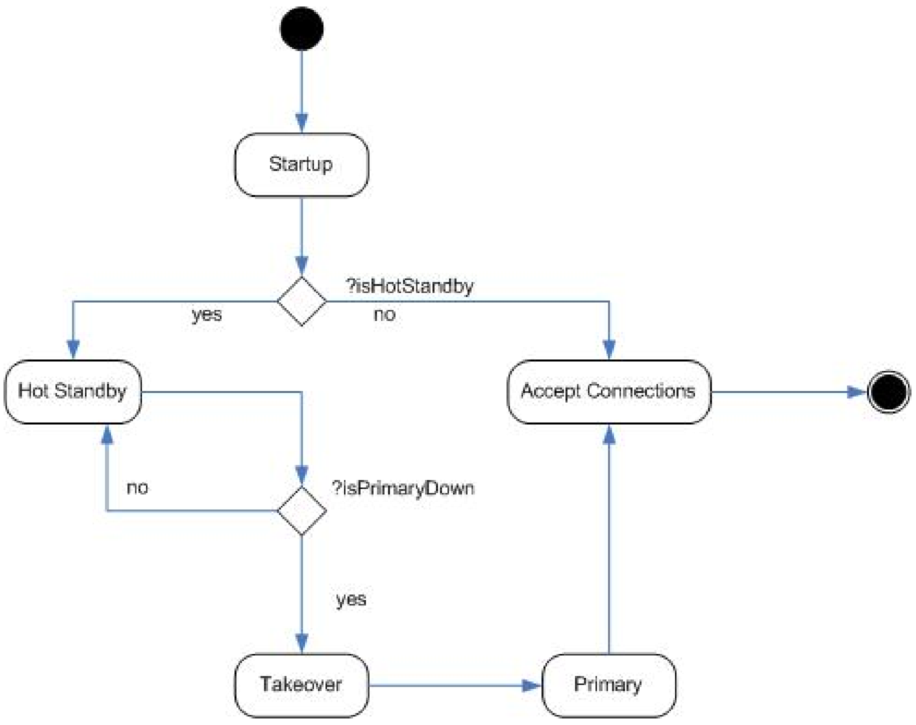


Figure 5.2: State Diagram: Network Server

5.9.2 Heartbeat

To enable fail-fast semantics messages telling the hot standby that the primary is still alive is needed. The primary ships messages saying ”I'm Alive” is sent to the hot standby at fixed intervals of time. If no heartbeat messages have been received by the hot standby within the allowed time[1] the primary is assumed down and takeover is engaged.

5.9.3 Normal Execution

During normal execution, the network server is unchanged with an exception of the hearbeats sent and received like mentioned above.

5.9.4 Takeover

When the primary is detected as down a takeover is needed. The takeover phase consists mainly of signalling the hot standby controller to commense takeover and abort all running transactions and then finally accept incoming connections when the hot standby controller is ready.

5.9.5 Catchup

If the hot standby has for some reason gone down, either from when it was primary or hot standby some synchronization between the databases is needed. The primary might be changed since the last time the hot standby was up. The primary detects the arrived hot standby and asks it for the last changes in its local logical log. Everything done from that last change is then sent to the hot standby to be redone. When the local logical log on both machines match the databases go back to normal processing.

5.10 Changes to the SQL Layer

Some changes need to be made to the SQL Layer/Execution Module in Derby to employ our new logical log. However this should be limited to collecting the information needed for the logical log and to invoke the logical logger with these values.

[1]The default timeout value for heartbeat messages has been set to ten seconds.

Chapter 6

Implementation

To implement the suggested log format the log record was divided into two parts: One part containing the control information of the log record: The transaction id, LSN of the record along with the previous LSN for this transaction. The name of the affected table is also included. The other part contains the operation information: What type of operation this is along with both the before- and after image of the record being changed. In addition to the logical log and the logger mechanism a hot standby controller is needed to keep track of the hot standby states, e.g. Is the hot standby alive? Am I hot standby? Finally the Derby network server must be changed according to the design choices mentioned in the previous chapter.

6.1 Network Server

When the derby database is to be used as a standalone DBMS in client/server mode and not in the standard embedded mode the network server is needed. The network server is responsible for getting jdbc connections and then forward the statements to the embedded derby database. How the network server interacts as an intermediate between the clients and internal databases can be seen in Figure 5.1 in chapter 5. Because of the network server implementation the invocations between the network server and the derby core are one-way. The network server can invoke methods inside the derby core, but the network server is implemented as protected and hence cannot be invoked from the outside. Due to this and the network server being responsible for client connections, the network server is a natural place to implement the hot standby mechanisms needed. The drda package is very large, so for simplicity only the affected classes are shown in Figure 6.1. The communication between the primary and hot standby network servers is implemented as a logical client and logical server much in the same way as inside the derby core.

6.1.1 Initial Startup

When the network server is started a NetworkServerControlImpl is started where the arguments provided by the invoker are processed and the appropriate actions are taken. The arguments for the network server are added in Appendix B. When

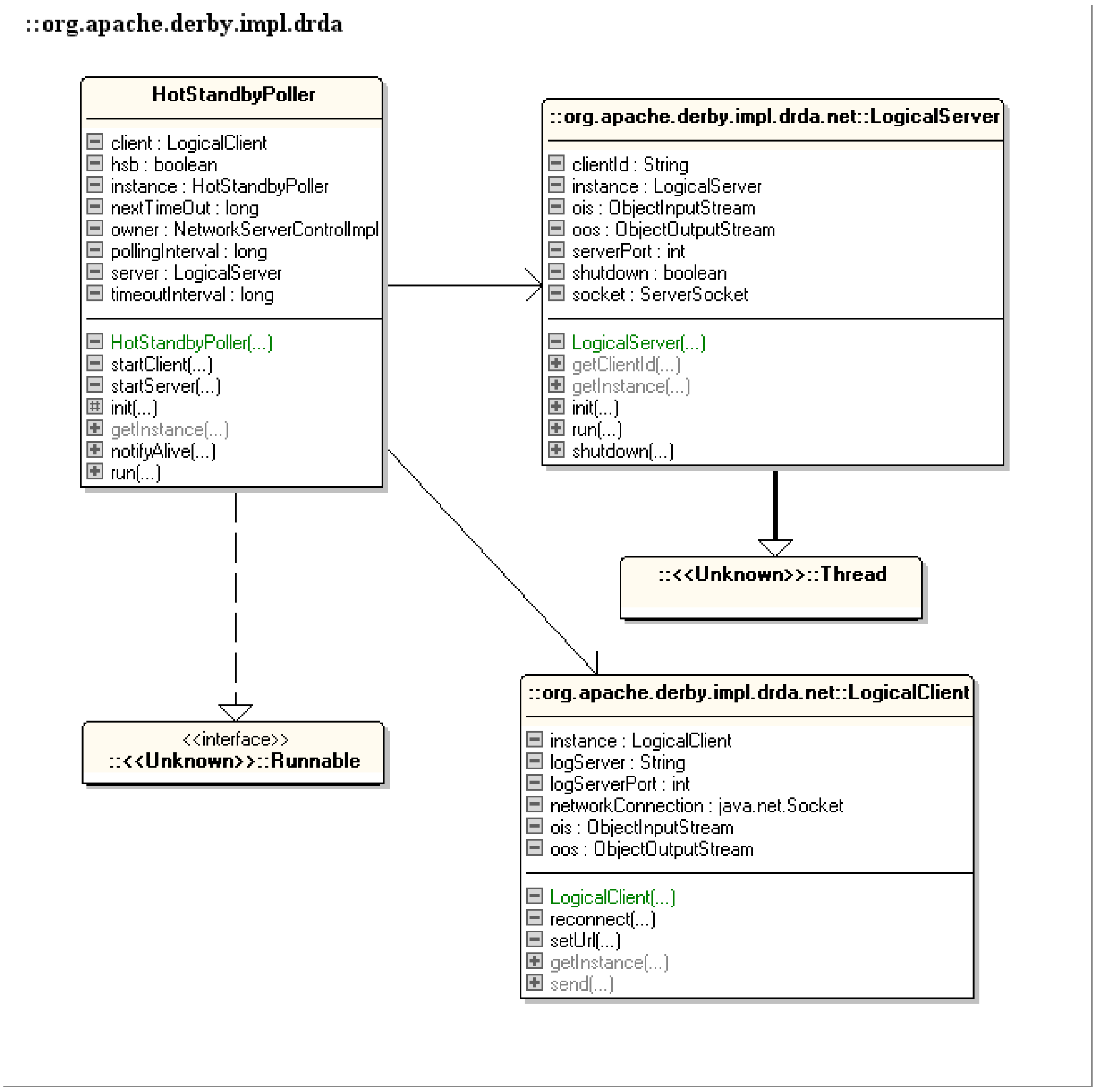


Figure 6.1: org.apache.impl.services.drda

the network server is started the hot standby controller and hot standby poller is initialized in the correct hot standby mode. When these have been initialized the network server is halted for as long as the database is hot standby using:

```
while (true){
        if (! IS_HOT_STANDBY)
                break;
        Thread.sleep(5000);
        //NOOP - wait to become primary
}
```

When the network server is set to be primary, the loop is broken and it resumes its normal execution and accepts client connections. In this way the transitions are null → hot standby → primary → null as described in the design goals.

6.1.2 I'm Alive and Takeover

During normal execution the added responsibility of the network server is mainly to send and receive heartbeat messages. The hot standby poller responsible for these messages has the same transitions and states as the network server itself, namely null → hot standby → primary → null.

Poller in Hot Standby Mode

When the hot standby poller is started in hot standby mode it goes into a simple waiting algorithm checking for timeouts. A count-down from ten seconds is initiated, if the countdown reaches zero the primary database is assumed to be dead and the correct measures need to be taken. Every time an I'm alive message is received the counter is reset. However, if the timeout occurs the hot standby mode is reversed to primary mode and the logical server is asked for the url of the old primary. This url is then passed to the hot standby controller, then the NetworkServerControlImpl's hot standby flag is finally changed to resume the execution of the network server and accept client connections. The logical server inside the drda.net package is shut down and the logical client is started to enable it to send I'm alive messages as soon as the new hot standby database is ready to reconnect.

Poller in Primary Mode

When the hot standby poller is initiated in primary mode it attempts to send an I'm alive package to the hot standby and receive an acknowledgment. If the connection is broken the poller will retry until the attempt is succesful. When a connection is made or broken the hot standby controller is notified that the hot standby is alive or dead. Then another I'm alive message is sent every second for as long as the duration of the server.

6.2 The Logical Log

The logical log is created as a linked list of log records. The logical log has references to the first and the last log record while the records themselves are interconnected. When a new log record is added it is checked that it's LSN is greater than the last LSN recorded in the log to ensure that the log records are not duplicated and that the log records are recorded in strict sequence. An illustration of a small logical log is shown in Figure 6.2. Simple methods to traverse the logical log are created and new log elements can be added either one by one as done with the logical logger or in batches as done by the log receiver. To avoid the log becoming too big and depleting the system memory the log is implemented as a circular buffer with a maximum size of 20.000. If the log is "full" the first element is removed and the next element is added to the end of the log.

Since the logical log acts as a buffer between the logical logger and the log shipper it is important that some care is taken when the log is transmitted to the log receiver. When the log is sent it is also purged to free needed stack space. To avoid race conditions where log records are added to the log after the log has been sent, but before it is purged, it is important to create a mechanism to avoid the problem entirely. When the log is to be sent it is first purged by invoking the flush method. This clones the entire log and returns it to the invoker and then resets and empties the log. The log to be sent is now extracted while the log itself is reset and ready to add more log records.

To avoid race conditions some thread synchronization is required. An example of a possible and fatal race condition can be describes as follows:

1. Transaction *a* gets an lsn and start the logical logging of the current operation.
2. Transaction *b* gets a new lsn and finishes its logging before *a*.
3. The logical operation of *b* is written to the log before the *a*. When *a* tries to add its log record to the log an exception is thrown due to *a*'s lsn being smaller than the last one recorded in the log and it is therefore discarded.
4. The logical log missed one or more log records and the log is left in an inconsistent state.

The logical log also contains some methods to help traverse and split the log as needed by the hot standby controller and log shipper/receiver. The split() method returns the subset of the logical log from the lsn provided and to the end, while the rewind() method returns a log element based on the provided lsn.

The log can easily be written to- and read from disk with the static methods *readFromDisk()* and *flushToDisk*. The log is then written to logicallog/logical.log

The interface for the operation types and images can be seen in Figure 6.3. The implementation of the log records, the whole log and the operations can be seen in Figure 6.4 and Figure 6.5.

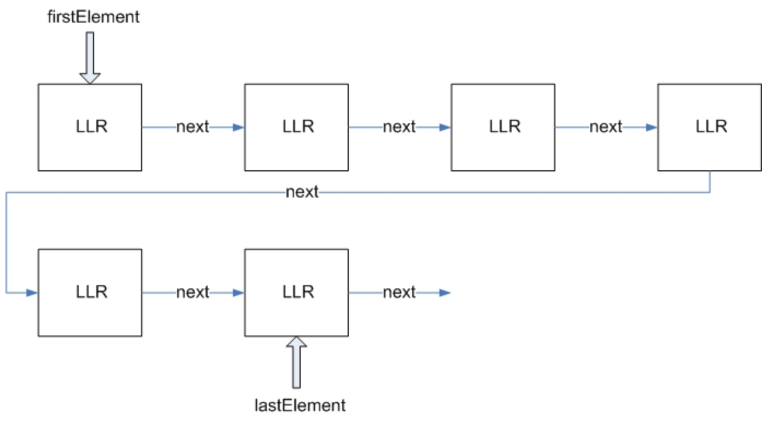


Figure 6.2: Logical Log Implementation

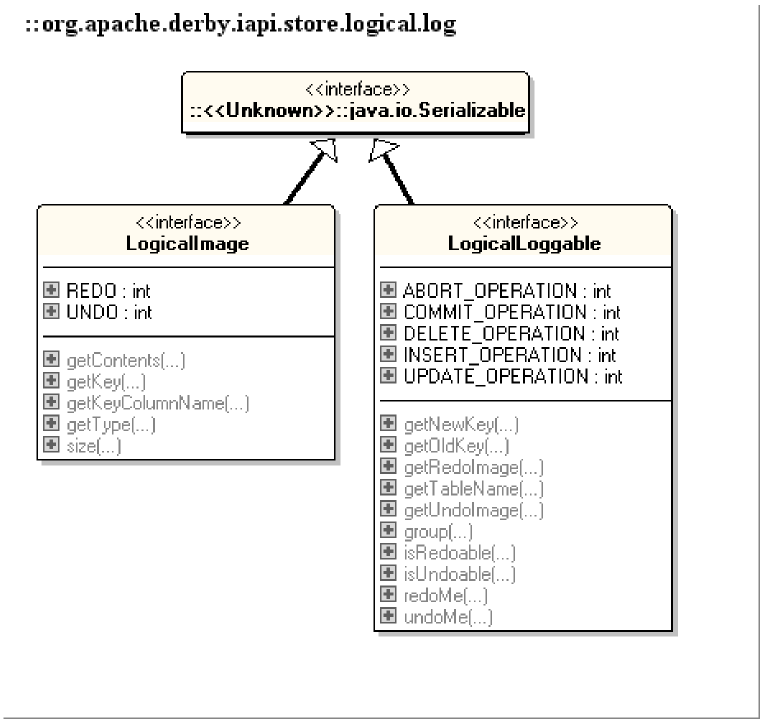


Figure 6.3: org.apache.iapi.store.logical.log

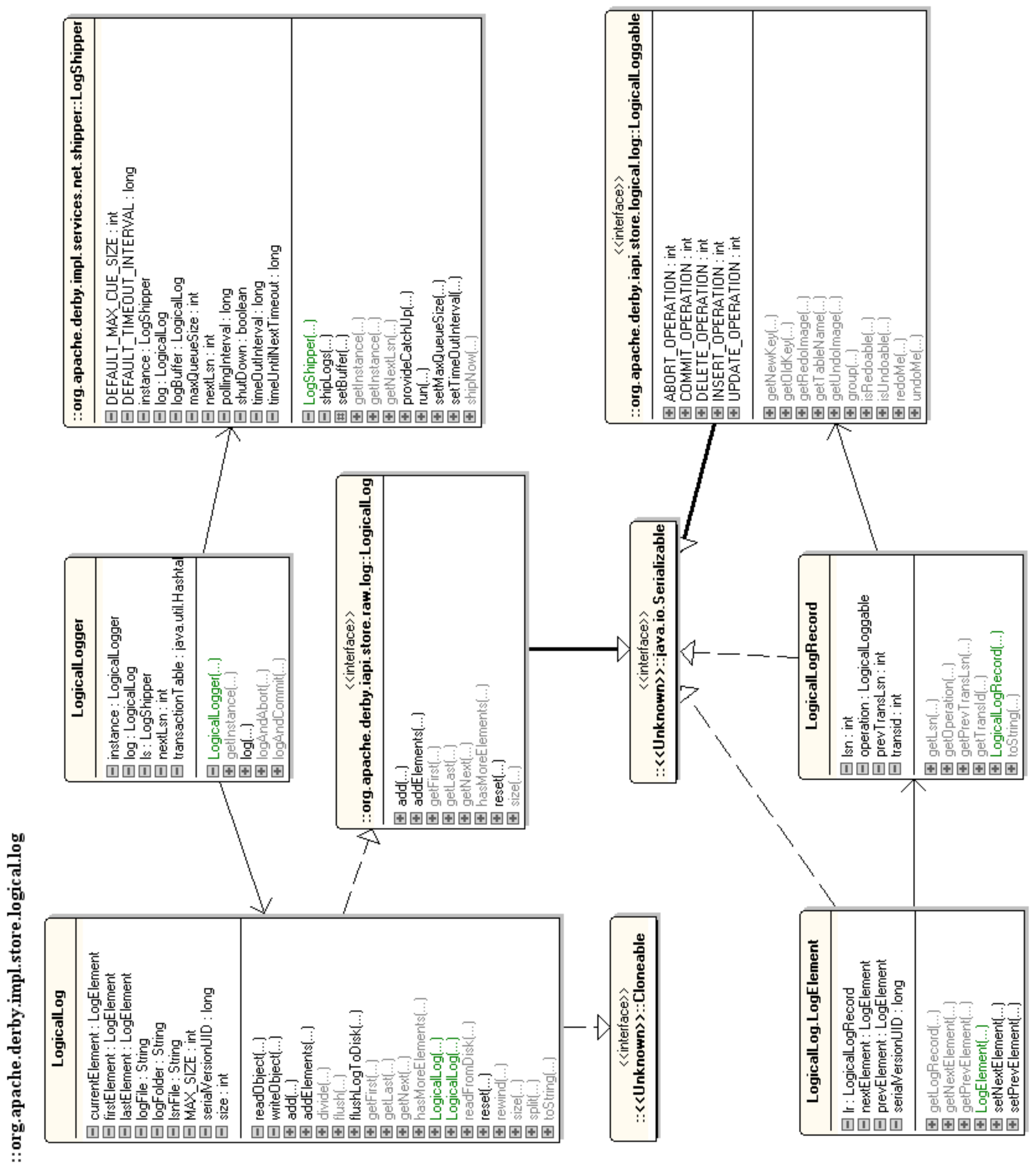


Figure 6.4: org.apache.impl.store.logical.log

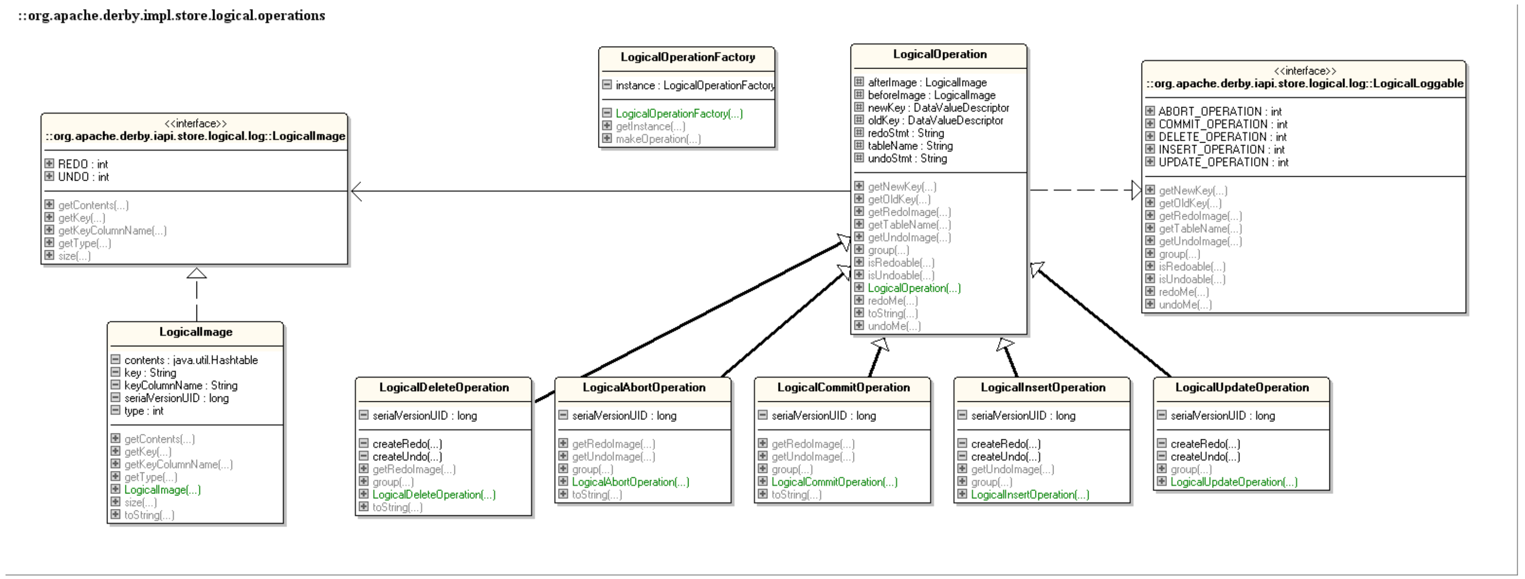


Figure 6.5: org.apache.impl.store.logical.operations

6.3 The Logical Logger

The logical logger is the class responsible for creating logical log records and adding these to the logical log. The old and new images of the record affected are used to create logical images that are added to the logical operation corresponding to the operation in progress (e.g. insert, update or delete). This operation is then added to a new log record along with the control information mentioned earlier. Since the logical log is to be shared with another node it is important that the primary and backup node agree on a LSN numbering convention. This can be done in two ways:

- The logical logger asks for the next LSN number from the receiver on startup
- The logger employs its own internal LSN numbers and these are changed when they arrive at the receiver before they are added to the log there. However, this approach removes the possibility for the logical log to be used for recovery since the numbering of the distributed log no longer match the ones recorded by the sender.

In our project the first approach is chosen. When the LSN is received by the logical logger it begins its normal operations. When the Derby database closes the connection to the log receiver the latest recorded LSN is written on the receiver and shipped to the database when it reconnects. However, if the system would consist of more than one primary database shipping log records to one hot standby and the logical log is used for replication only, the second option would be more preferable since the cost of synchronizing the LSNs between the databases would increase greatly for each new database added to the system.

The Logical Logger is created as a global singleton[1] to avoid race conditions and synchronization issues. The global derby system share this logical logger and its LSN numbering scheme. If the services provided by this global instance is to avoid race conditions they need to be synchronized. To avoid these race conditions under heavy load the log, logAndCommit and logAndAbort methods are synchronized.

To provide one- and two-safe replication the logical logger provides three services to the Derby system, the normal logging service along with methods for committing and aborting the log. When the normal log method is invoked a new logical log record is created with the LogicalLogFactory and added to the logical log. The logAndCommit and logAndAbort methods however depend on the mode set. If the mode is set to *ONE_SAFE* the commit log records are added to the logical log as if the normal log method was invoked. If the mode is set to *TWO_SAFE* however the system employs a two-safe-like shipment where a simple implementation of the two phase commit protocol is used. The commit operation is put on halt until either the 2pc messages are passed and the log is received successfully or an error has occurred. If an error occurs during two-safe

[1]Only one instance of the LogicalLogger can be created

commit an abort-log record is written instead and the transaction will be aborted instead. How these modes affect the log shipments are shown in the Log Shipper section and how they affect commits and aborts are shown in GenericLanguage-ConnectionContext.

6.4 Hot Standby Controller

To ensure that the log elements received by the hot standby are correctly redone and to ensure the ACID properties of the database scheme[2] some control is needed and the hot standby controller is created for this purpose.

The hot standby controller has two main purposes: To provide the derby system with the states of the hot standby system and to provide the hot standby access to the embedded database and to redo/undo log records, depending on aborts or commits.

6.4.1 Hot Standby States

The different states saved in the hot standby controller are:

- **hotStandbyAlive** The hotStandbyAlive is a boolean telling if the hot standby is alive or not. It is only needed when the database is in primary mode and used by the LogShipper to check if the hot standby is alive before an attempt to ship logs are made.
- **hotStandbyUrl** The url to the hot standby database. This is set by the network server on startup as it is passed as an argument to the network server when started if the database is started in primary mode. However, if the database is started in hot standby mode, the hotStandbyUrl is set to the url of the connecting primary. If the primary should go down the hot standby will become primary and know where the new hot standby will be when it comes back online.
- **isInHotStandbyMode** This is a boolean telling the system if this database is in hot standby mode or not. If isInHotStandbyMode is set to true the logical logger will be disabled and no log shipments will be attempted.
- **replicationMode** To support one-safe and two-safe replication as described earlier the hot standby controlloer has a mode that can be set to either *ONE_SAFE* or *TWO_SAFE*. These modes will then be read by the logical logger on commits and aborts and the correct approach is taken as described earlier.

6.4.2 Hot Standby Support

The states mentioned above are very useful to the hot standby scheme, however the main responsibility of the hot standby controller is to provide the hot standby database the mechanisms needed to redo or undo the received log files depending

[2]The database scheme meaning the primary node and hot standby node combined.

on the transaction being committed or aborted.

The hot standby controller has an internal, very simple transaction table implemented by a Hashtable containing the transaction id and the lsn of the last log record done by this transaction. It also has a method to connect to an internal, embedded database using the Database class. The Database class is responsible for fetching jdbc connections to a given database. When a logical log is received by the log receiver it is forwarded to the hot standby controller where it is put in a log buffer. If the buffer is full a wait-message is returned to the primary database to signal that it has to retry in a little while due to the full log buffer. The buffer is needed to avoid the hot standby falling too far behind the primary. If the primary is not put on wait the memory on the hot standby would fill very quickly and the system would eventually crash. When the log is added to the buffer it is eventually reset and scanned from the start. If the transaction is not already in the transaction table it and the lsn of the log record is added to the transaction table. If the transaction table already contains the transaction it is updated with the lsn of the log record. Finally if the operation is redoable it is redone. If the log record is a commit record the transaction is simply removed from the transaction table and the transaction is considered committed. However if the log record is an abort record the transaction is rolled back using the private method abortTransaction. This transaction scans the log from the last lsn recorded in the transaction table and traverses the log backwards, undoing the records as it goes. When all the log records have been undone the transaction is removed from the transaction table and no further actions will be redone or undone on its behalf.

If a takeover is needed due to the primary going down the hot standby controller is notified of this through the setHotStandbyMode method. Before the mode is set a clean-up is done. The clean-up goes through the transaction table and aborts all the transactions recorded there in the same manner as before.

The hot standby controller service is shown in Figure 6.6

6.5 Communication

To be able to send and receive logical log records the log shipper and log receiver was created. The connection between the two are implemented by a TCP socket that is kept alive for as long as the database is running. To simplify and hide the communication implementation from the rest of the database a request-response protocol is created. The communication is created as a service in the service layer of Derby in the packages
org.apache.derby.impl.services.net,
org.apache.derby.impl.services.net.receiver and
org.apache.derby.impl.services.net.shipper.

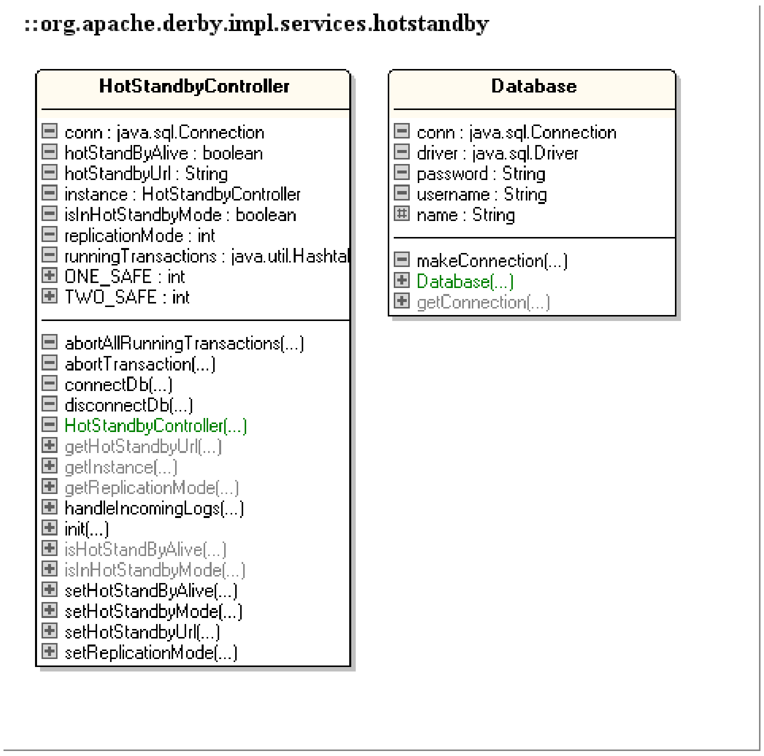


Figure 6.6: org.apache.impl.services.hotstandby

6.5.1 Logical Client and Server

The request-reply protocol needed to exchange both logs and messages between the primary and the hot standby databases is implemented in the *LogicalClient* and *LogicalServer* . When started, the logical server waits for connections[3]. When the Logical Client an attempt to connect to the server is done. If the client is unable to connect it will wait for a random amount between 0 and 10 seconds before a reconnection is attempted. When the connection is complete it is kept up for the duration of the server. If the connection is disconnected a reconnect is issued until the connection is back up.

To keep the communication protocol simple, well defined messages have been defined. The client is implemented as a singleton, with one available function, send, to keep it simple and easy to use. The send-method has one input-parameter, a NetworkPayload and it returns a NetworkPayload. The NetworkPayload is a simple class containing a type and the object containing the payload to be sent or retrieved. The different types of payloads available are:

- **Log** The most important type is the log. The payload contains a LogicalLog object.
- **ACK** The ack is an acknowledgement that the request went well. The

[3]The logical server inside the Derby core listens for connections on port 12345 and on port 12346 in the Network Server described in section 6.1

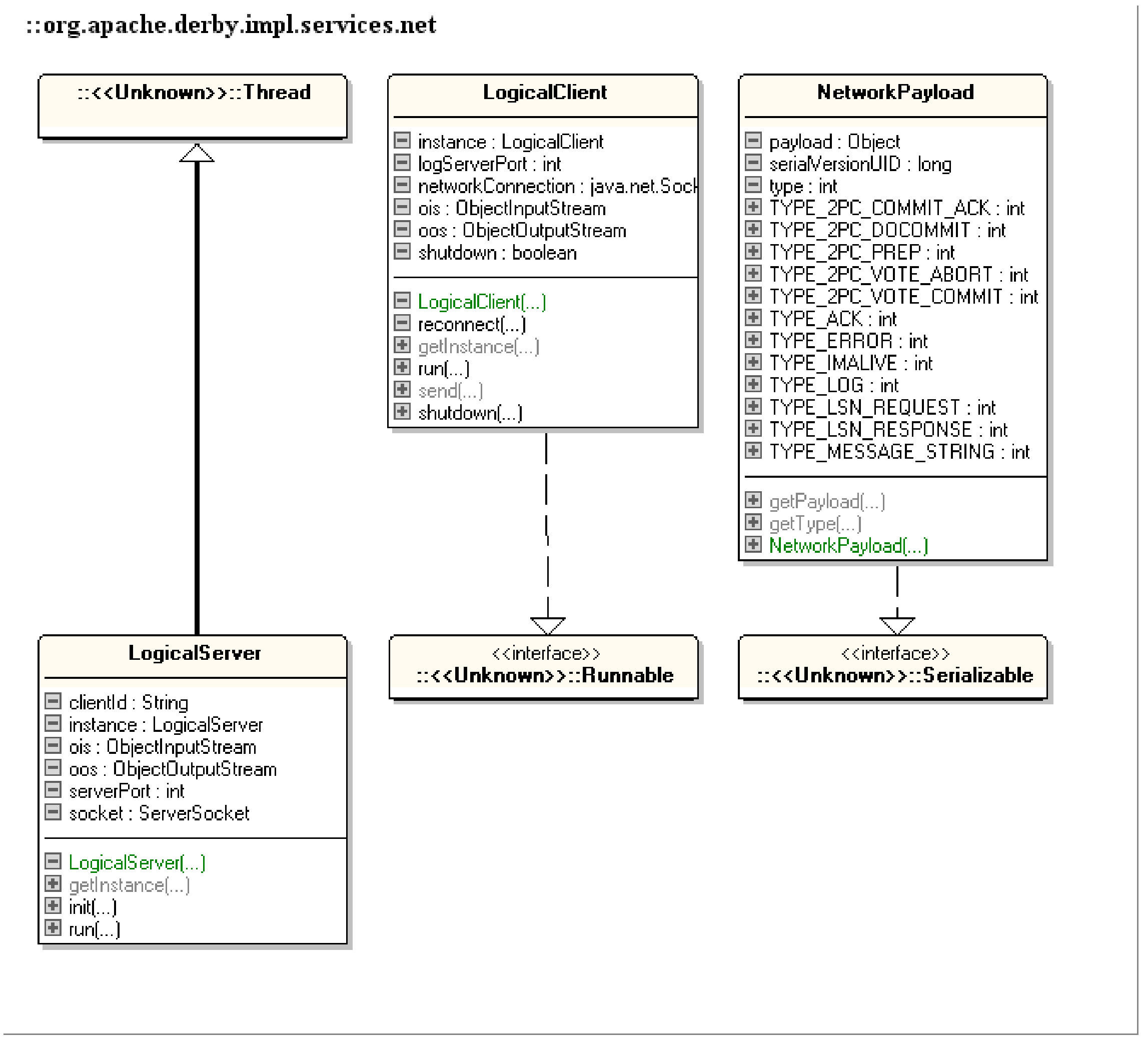


Figure 6.7: org.apache.impl.services.net

payload contains a textual representation of what was acknowledged, useful for logging and debugging purposes.

- **2PC Prepare** A simple two-phase-commit prepare message.
- **2PC Vote Commit** A vote to commit, a positive answer to a 2PC prepare.
- **2PC Vote Abort** A vote to abort, a negative answer to a 2PC prepare.
- **2PC Do Commit** A message telling the hot standby that the vote is in and it should commit now.
- **2PC Commit ACK** An acknowledgment that the 2PC commit is done. When this message is received by the primary it is allowed to commit as well.
- **I'm Alive** I'm alive messages are used as heartbeats in the network server. The payload contains a String with the unix timestamp from when the message was sent. This timestamp is for debugging purposes only though and is not used. The I'm alive-messages and protocol is more thoroughly

described in section 6.1.

- **Message String** The messages string is a type reserved for simple textual messages. The message type is not in use.
- **LSN Request** An LSN request is used when the system is first started to check if the hot standby is up to date. It is also used when a hot standby needs to be caught up.
- **LSN Response** The LSN response is the latest lsn from the database.
- **Error** The error type is used to notify the requester that something went wrong along the way and the message was not sent or received correctly.
- **Wait** The wait message is used by the hot standby to tell the primary database to slow down its execution in order to help the hot standby keeping up.

The LogicalClient resets its ObjectOutputStream, puts the NetworkPayload on the stream and flushes it. The NetworkPayload is then retrieved on the LogicalServer. Depending on the type mentioned above, the appropriate action is taken. Inside the derby core the log, lsn requests and 2PC requests are used. Incoming logs are handed to the log receiver, LSN and 2PC requests are handled directly by the LogicalServer. Finally an answer is composed and returned to the LogicalClient where it is eventually returned to the invoker of the send-method. The complete net-service can be seen in Figure 6.7

6.5.2 Log Shipper

The log shipper is responsible for the communication between the logical logger and the log receiver. The log shipper shares the logical log with the logical logger and in this way the logical log acts as a buffer between the logger and the shipper. Log records are sent to the log receiver using the Logical Client described above when one of the following conditions are met:

- **A timeout has been detected.** A timeout value can be set to the log shipper to tell it to ship the log on timeouts. This ensures that the log is sent even if there are long idle times with no operations in the primary datbase. By setting the timeout value to a short time interval less log records are prone to be lost if a crash occurs since logs are sent more often. However, a too short timeout value will make the system send log records more often and thereby wasting valuable network bandwidth and processing power.
- **The logical log size has exceeded a certain limit.** When the logical log size has exceeded a predefined limit the logical log is transmitted. The reason to send the log when it has exceeded a certain size is to ensure that it is not lost as with the timeout case above. Since the logical log is implemented and sent as an object however, it is important to not let the size of the logical log exceed the allocated stack size[4] or the java stack will overflow resulting in lost log records.

[4]The java stack view the log as a run-away recursion if it grows too large due to the repeated next()-invocation on every log element. The limit is found to be close to 3000 elements

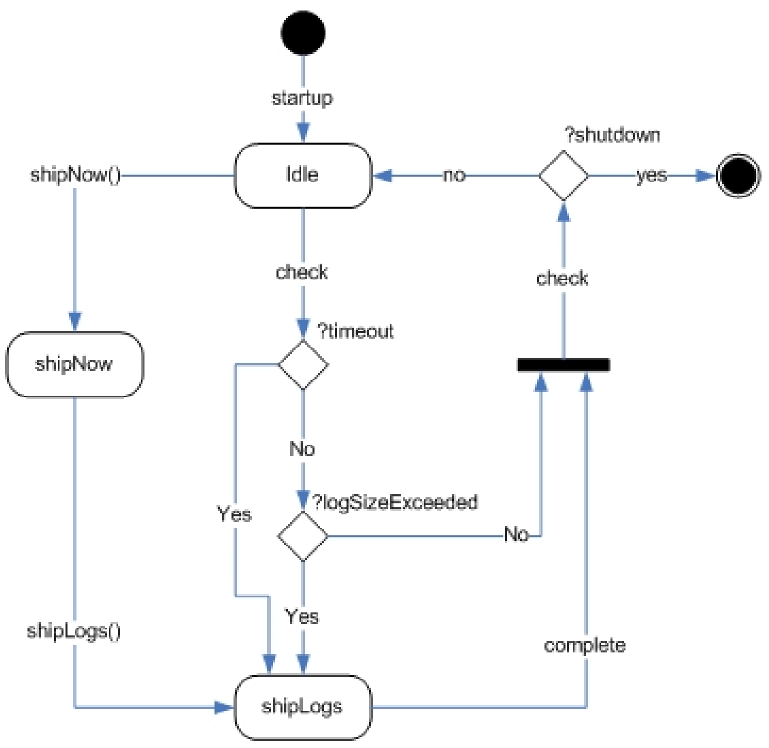


Figure 6.8: State Diagram for the Log Shipper

- **Forced sends due to two-safe commits triggered by the logical logger.** When the logical logger is set in two-safe mode the log records must be sent to the log receiver before their owner transactions can be allowed to commit. To enforce this policy a method, shipNow, has been created which attempts to ship all the logical logs including the commit record at once. If the transmission of the log records goes without any exceptions a *true* boolean value is returned to the logical logger and the transaction in turn which is then allowed to commit. However if an error occurs and an exception is thrown the shipNow method returns a *false* back to the logical logger. This means that the 2PC procedure has failed and the transaction needs to be aborted, not committed.

The log shipper goes into a loop when started which continually checks if any of the conditions above are met. If the log is determined to be sent the private method shipLogs is invoked and the logical log is sent. When a log is sent, the log object is written on an ObjectOuputStream and the stream is then flushed and reset. Finally the stream is emptied and is ready to be written to again. The stream remains open for the full duration of the connection. The different states of the shipper can be seen in Figure 6.8

6.5.3 Logical Catchup

The log shipper also contains the LSN of the next logical log record. When the log shipper is initially created this value is set to 0, but when the first LSN is requested, the log shipper asks the hot standby for the next LSN. When the next LSN has been received, the log shipper continues to use and increment this field if the lsn is the same on both databases. If the hot standby LSN is smaller than that of the primary a catchup is initiated as described below. If the hot standby LSN is greater than that of the primary the primary is obviously outdated and this is considered a critical error. The primary is shut down and the next time it is started as hot standby it opts the other database for a catchup. The next LSN is only requested once by the log shipper on startup and then every time a hot standby resurfaces, this is to provide catchup if needed.

In addition to just shipping the logs to the hot standby the log shipper also provides catchup for resurfacing hot standby databases. If the hot standby has been disconnected for some reason, either through a takeover or another disconnection issue, it needs to be caught up with the primary before normal execution can be continued. To avoid the database being halted while the catchup is in progress it is implemented in a separate thread in the LogicalCathcup class. The log is scanned for changes after the last remote lsn reported and all log records with a higher lsn than that are shipped to the HotStandby. The local logical log is read from disk, compared to the remote lsn reported and split into chunks of 3000 log records who are in turn sent to the HotStandby. This step is repeated until all of the log residing on the disk is shipped. This is needed as the log on disk can be changed while the catchup is in progress. Finally the service notifies the HotStandbyController that the HotStandby is now alive again and normal log shipping is resumed.

The implementation of the log shipper package is shown in Figure 6.9.

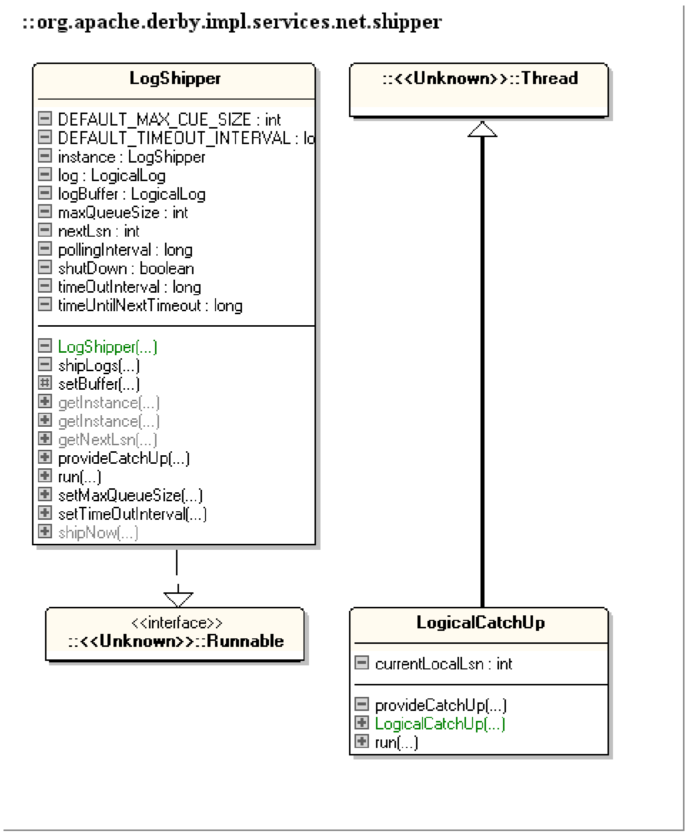


Figure 6.9: org.apache.impl.services.net.shipper

6.5.4 Log Receiver

The log receiver is created as a simple process which receives, forwards and stores the logical log. When the log is received it is added to the active log, handed over to the hot standby controller for processing and finally it is flushed to disk. To improve the response times of the hot standby scheme an early answer is returned to the primary when the log is received. Then the log is handed over to the log receiver. The states and transitions for the logical server and log receiver is shown in Figure 6.10 while the implementation diagram is shown in Figure 6.11

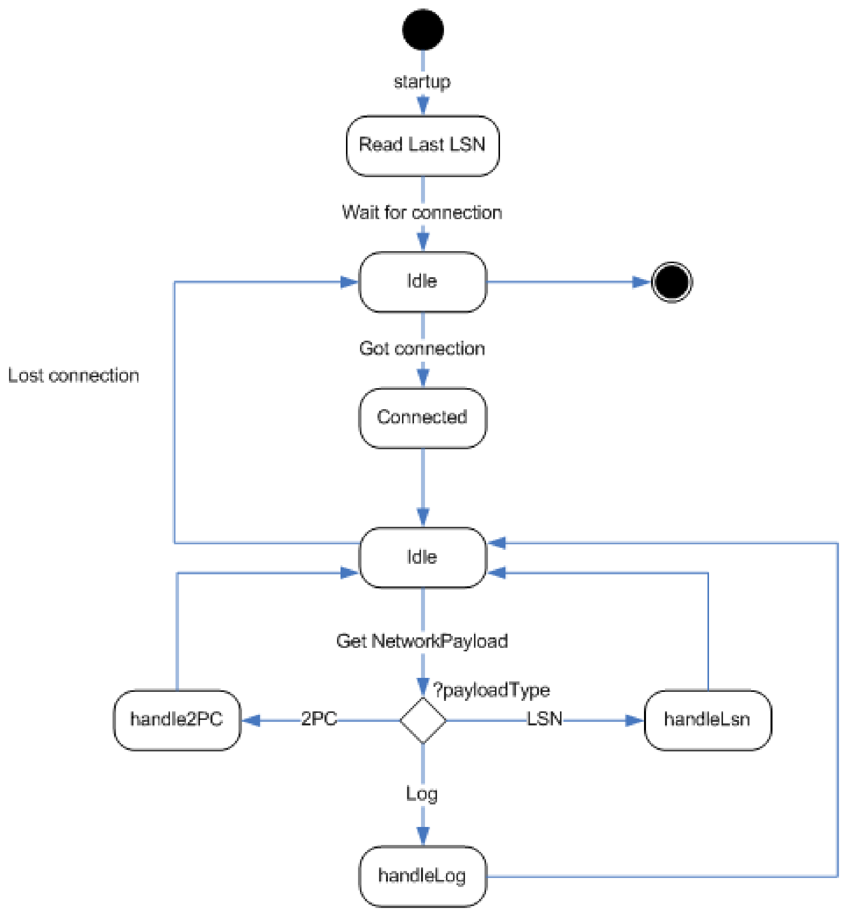


Figure 6.10: State Diagram for the Logical Server

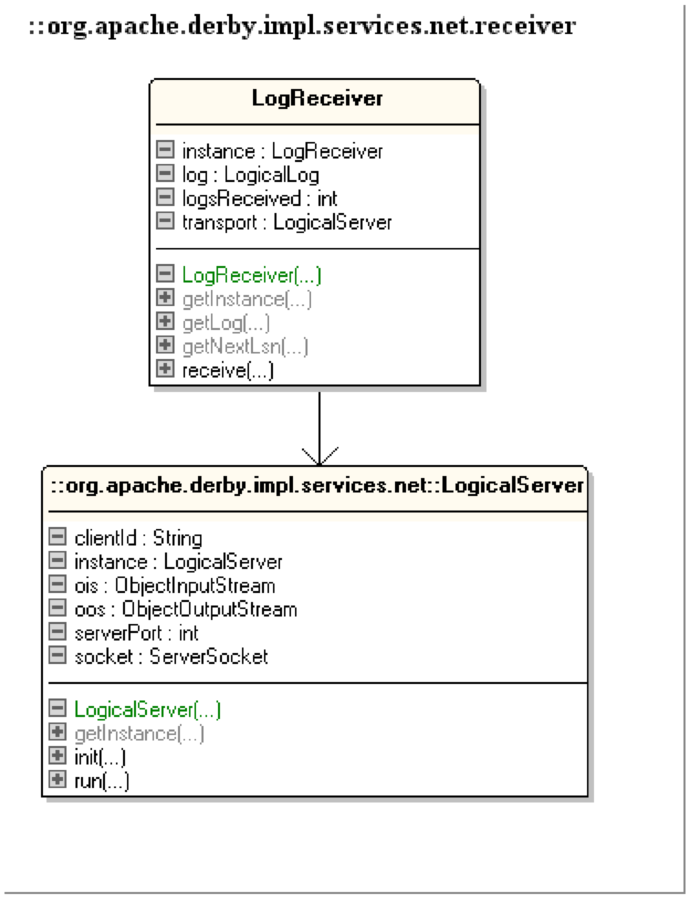


Figure 6.11: org.apache.impl.services.net.receiver

6.6 Changes to the SQL Layer

To make Derby use the implemented logical log some changes had to be made to the SQL-layer as well. The changes done to Derby is limited to the classes org.apache.derby.impl.sql.execute.RowChangerImpl and org.apache.derby.impl.sql.conn.GenericLanguageConnectionContext

6.6.1 RowChangerImpl

After some investigation the class RowChangerImpl residing in the package org.apache.derby.impl.sql.execute was found as the most appropriate place to implement the log. The methods insertRow, deleteRow and updateRow contained all the information needed for our new log:

- **insertRow**: When a row is inserted the full row is provided with this method. To find the new key (if any) is done by consulting the data dictionary to find out which column in this row is the key. The name of the table this record belongs to is also found by checking the built-in dictionary in Derby. The before image is not needed when creating a new row and is therefore not included in the new logical insert operation.
- **updateRow**: The most complex of the three operations is the update operation. When an update operation is performed the old and new rows are provided to the method. However, these only contain the changed values and a reference to what page it is stored on the disk. We have decided only to use the changed values in our logical log and therefore we only need the changed values, but the old and new key are still required. To find these we look up the key column from the dictionary as usual. Using this key information we fetch the key column from the database using the HeapConglomerateController. If the new key is provided with the updateRow method invocation we now have both the old and the new key. If not the key is not changed and the new key is the same as the old one.
- **deleteRow**: The deleteRow method is very similar to the insertRow method. The difference here is that the entire row is fetched from the database using the HeapConglomerateController before the delete is executed and put in the before image of the operation. The after image is set to null.

6.6.2 GenericLanguageConnectionContext

In RowChangerImpl all the operations regarding changes to records are recorded. However we also need to create logical log records for commit and abort operations. All transactions that finally either commits or aborts do this through the doCommit and doRollback methods in GenericLanguageConnectionContext. Whenever a transaction is to commit the id of the running transaction is fetched and provided in a logAndCommit or logAndAbort method invocation to the logical logger. The logical logger then returns the result of the commit in terms of the log records being shipped successfully or not. If the logical logger is set in one-safe mode logAndCommit or logAndAbort always returns a true value and the transaction is allowed to commit. If the mode is set to two-safe however the logAndCommit or logAndAbort invocation is blocking meaning that the transaction needs to wait for the outcome before they are allowed to commit.

6.7 A Simple Transaction Example

A simple transaction with one update and a commit can be seen in Figure 6.12. For simplicity the Derby transaction system as a whole has been encapsulated in the *Derby Transaction* object. When an update is done to one or more rows the operation is done in *RowChangerImpl* as usual and then logged using the LogicalLogger. When the transaction decides to commit the doCommit method within *GenericLanguageConnectionContext* is invoked. Since two-safe is the chosen mode the transaction is put to wait while the *LogicalLogger* forces the log from *LogShipper* to *LogReceiver*. Depending upon the answer from the logical logger the transaction is able to commit. If one-safe replication is selected however the transaction would commit immediately after the logAndCommit invocation as shown in Figure 6.13.

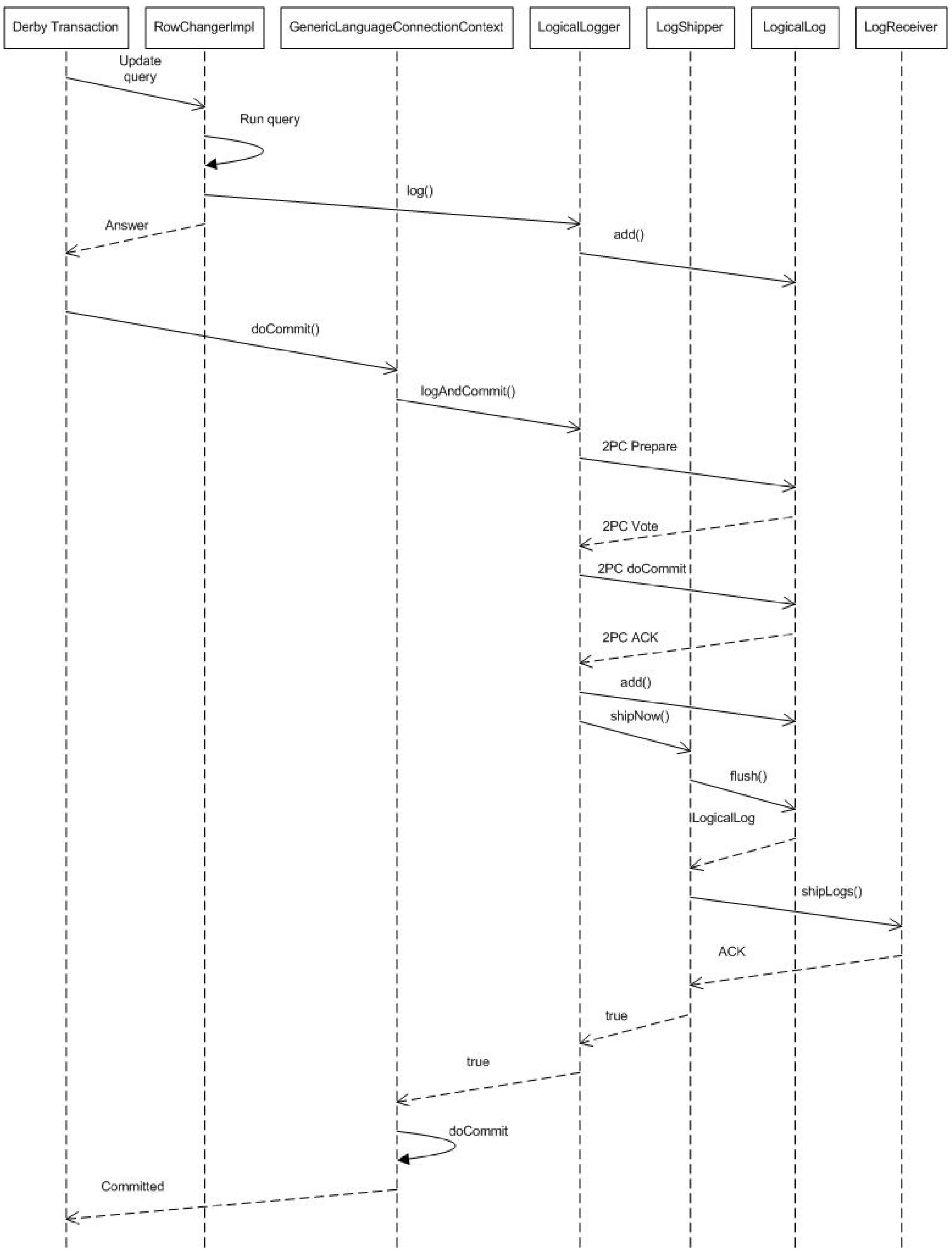


Figure 6.12: Simple Transaction with two-safe commit

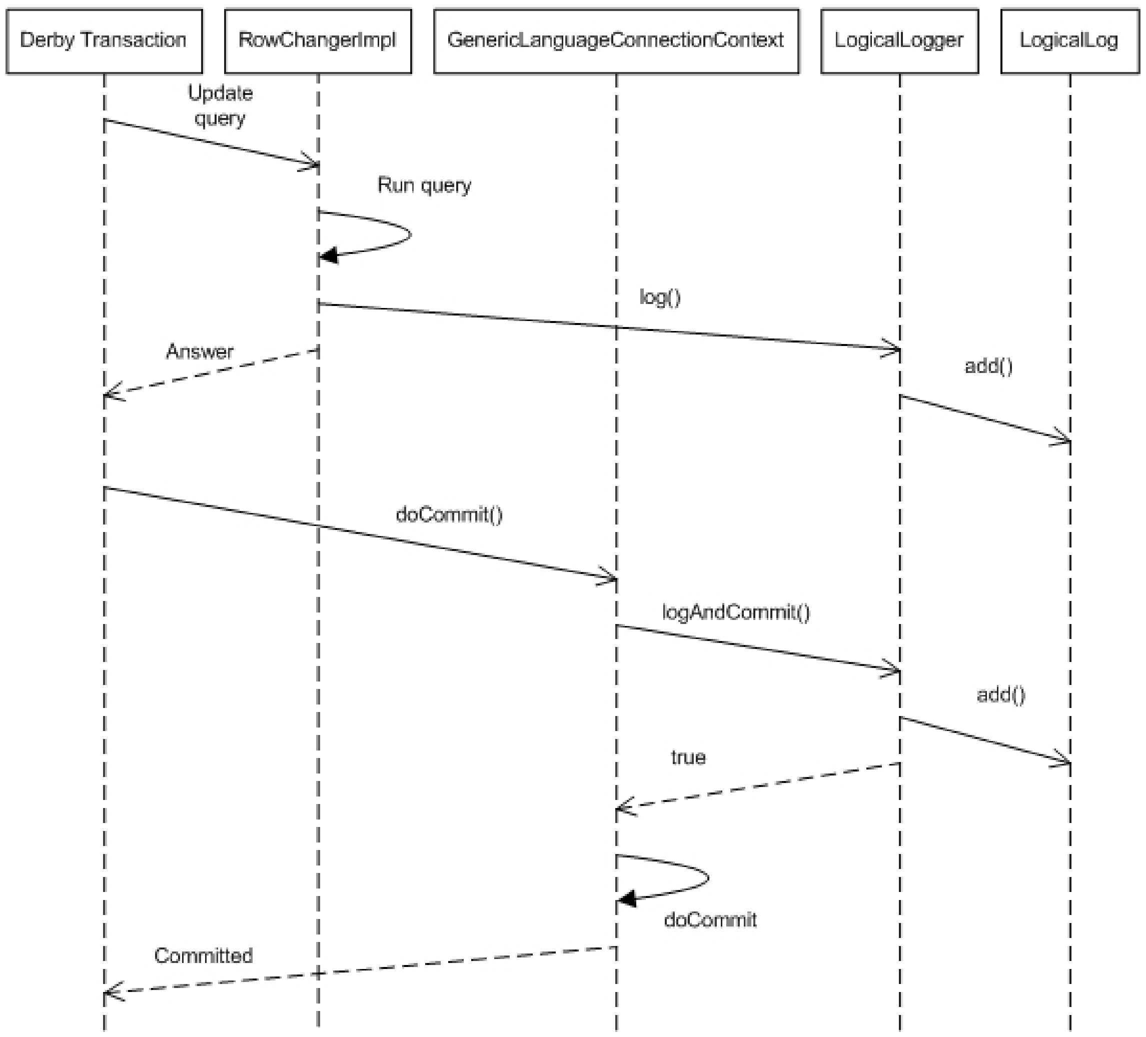


Figure 6.13: Simple Transaction with one-safe commit

Chapter 7

Measured Results

To compare the new hot standby scheme to the original Derby network server a benchmark is needed. A comparison of the original system and the new system using both one-safe and two-safe replication was done using a TPC-B like benchmark.

7.1 The TPC-B Benchmark

The TPC-B benchmark was created by the Transaction Processing Performance Council (TPC) to exercise database components in update-intensive database services such as:[14]

- Significant disk input/output
- Moderate system and application execution time
- Transaction integrity

Transactions are created and run for a limited time and the throughput of the system is measured in Transactions per Second (TPS). The database system is set to simulate a simple bank with one or more branches. Each branch has its own tellers, customers and accounts. Each transaction makes an update to an account, teller and branch before writing the operation to a history table. To see the full implementation of the TPC-B like benchmark used in this project see Appendix A.

7.2 Benchmarking Architecture

The Derby DBMS can, as stated earlier, be used in two different modes: Embedded and Server. To enable multiple clients connecting to the DBMS at once the server approach was selected. With the server approach the network utilization also becomes a concern due to the log shipper and receiver sharing their communication line with the server interface. This can of course be avoided by providing a dedicated line for the log shipper and receiver. However, as we will see, the communication line is not the biggest concern, at least not for performance reasons.

The computers used for both the primary and hot standby servers are two Pentium 4 3GHz computers, both with 1GB of RAM and running Ubuntu Linux. The network line between the server/client and the server/log server is a 100Mb full-duplex ethernet line on an unprotected network. Due to the server only having one network card the network line between the primary/client and primary/hot standby is shared. The clients are run on a MacBook 2 GHz with 1GB RAM running Mac OS X 10.4.8. All the computers use Suns Java Developers Kit (JDK) 1.5.0. The complete architecture for the benchmark is shown below in Figure 7.1.

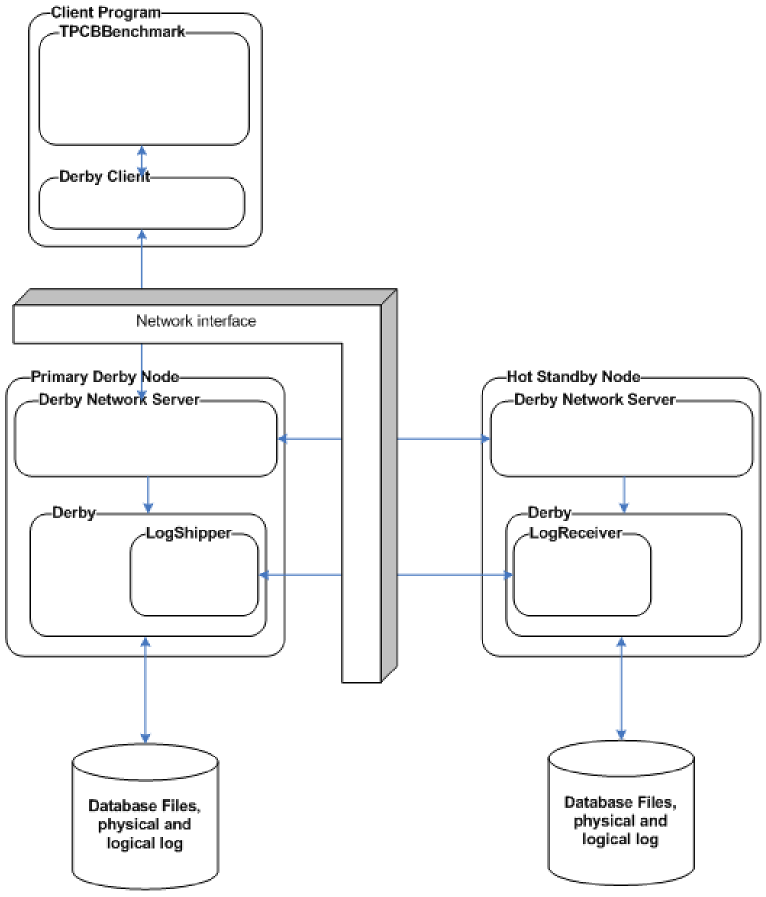


Figure 7.1: TPC-B Benchmark Architecture

Transactions are running back-to-back, i.e. clients send new transactions to the server as soon as they receive replies on the previous ones. To achieve multiple client connections each client is implemented as a thread, these clients share the same JVM[1]. The complete source code for the TPC-B Benchmark used is shown in Appendix A.

7.3 Results

The benchmark was run with one to nine concurrent clients for fifteen minutes each and the results were collected using *vmstat* for cpu utilization, *iostat* for disk utilization, *nload* for network utilization and the benchmark itself for throughput and response times measurements.

7.3.1 Throughput

The throughput achieved in the benchmark is shown in Figure 7.2. As we can clearly see the original Derby system is superior to both the one-safe and two-safe replication approaches. While this is no surprise due to the added complexity and functionality, the results show significant room for improvements. When using the original network server it is shown that when the system load goes from one to three clients there is a slight improvement in throughput. The reason for this is simple: Derby still has some idle times during executions which in turn could be used for additional transactions. However, the throughput is reduced when adding more clients.

The one-safe approach has a lower, but steady throughput rate. Still, if the load is removed from the hot-standby[2] the throughput is very close to the original system. Also if the benchmark is run from a client connected to a slower network line the throughput is also very close to the original. The hot standby cannot keep up the pace, it is just not as fast when redoing the sql statements as the primary is when it is doing its prepared statements. To understand why this happens take a closer look at the system implementation described in section 6.4 and section 6.5. If the hot standby redo routine could be improved by either utilizing prepared statements or some other means to redo the logical logs this throughput penalty could be avoided. Due to the logical logger having a single-point-of-entry it also becomes a bottleneck when more than one transaction tries to log simultaneously. If each transaction had its own logical logger this could also have been avoided, though some care must be taken in synchronizing these logical loggers to create log elements on a shared log with a strict lsn numbering scheme.

As can be seen, the two-safe protocol, not surprisingly, suffer greatly in regards of throughput. While every transaction need to wait for the 2PC messages and the logical log to be shipped successfully before they leave the logical logger's

[1]All the clients are run from the same client machine
[2]The load meaning the redo of the received logs

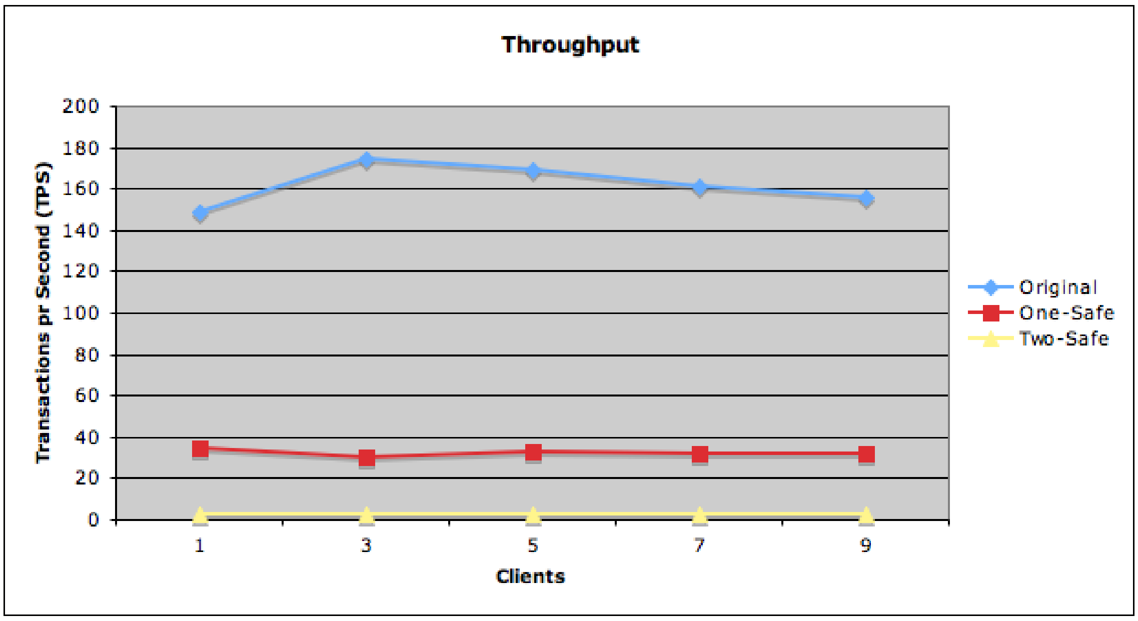


Figure 7.2: Average Throughput

logAndCommit-method the throughput is restricted by the log shipper and log receiver in addition to the logical logger. The severe penalty imposed to the system by the two-safe replication however is expected as this is a well-known problem with two-safe replication.[3, 5, 6]

7.3.2 Response Times

The average response times shown in Figure 7.3 coincides well with the throughput observed. While the total throughput of the system increases with the addition of additional clients to the system the response times are kept low. However, when the throughput stays constant or decreases the response times increase due to more transactions waiting for resources and service from the system.

As can be seen the original and one-safe approach performs rather well compared to the two-safe approach. The average response times stays below 300 ms for the one-safe replication and below 100 ms for the original Derby implementation. The one-safe approach has roughly the same response times as the original for few clients, but it does not scale so well when more clients are added. The reason for this is that the hot standby has trouble keeping up when the load is high. The two-safe approach however has more dramatic increase in response times due to the throughput being low and keeps decreasing for each client added to the system. The increase in average response times for the two-safe approach is almost linear and it is therefore easy to predict it to keep rising with the same factor for every added client. It is safe to say that the two-safe approach is not suited for systems having many connected clients and a generally high database load. However it is the only way to safely guarantee all the ACID properties for the system as a whole.

The maximum and minimum response times are also recorded and shown in Figure 7.4 and Figure 7.5. There is one surprise in that the two-safe approach has a lower maximum response time than both the original system and the one-safe approach. The reason for this might be that all the transactions are served as quickly as possible all the time, while in the one-safe approach it happens that the whole primary needs to be halted when the hot standby signals it to wait due to it having trouble to keep up the pace. However the majority of the transactions have a response time close to the maximum recorded in the two-safe approach. The one-safe and two-safe have higher maximum response times, but most of the transactions have low response times. The maximum response times gotten from the original Derby network server seem coincidental as the average response times are so low.

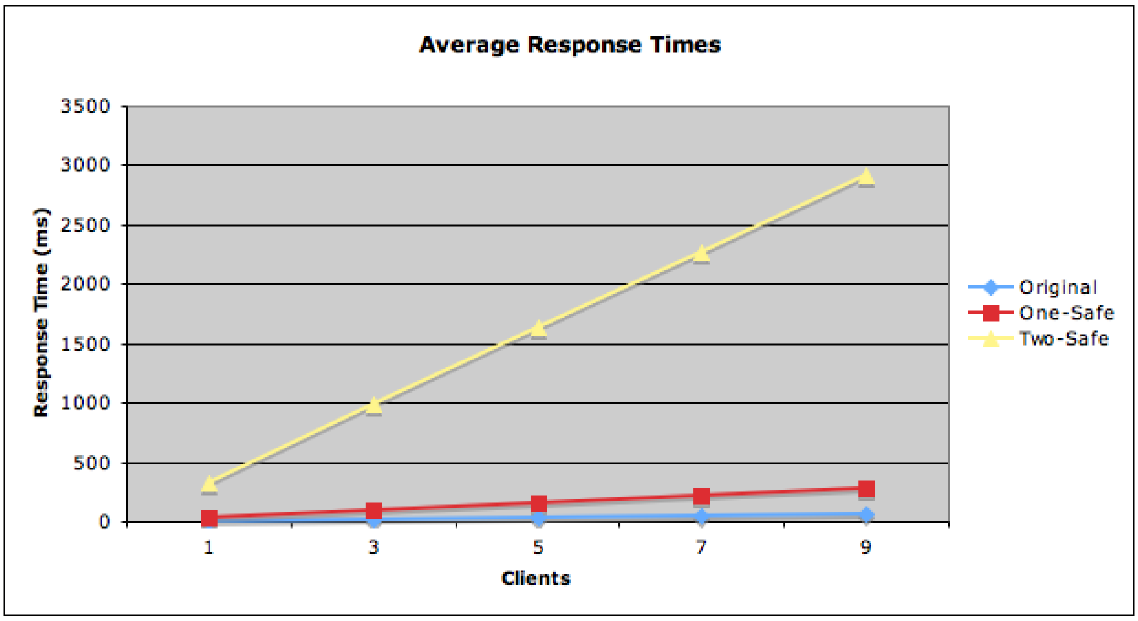


Figure 7.3: Average Response Times

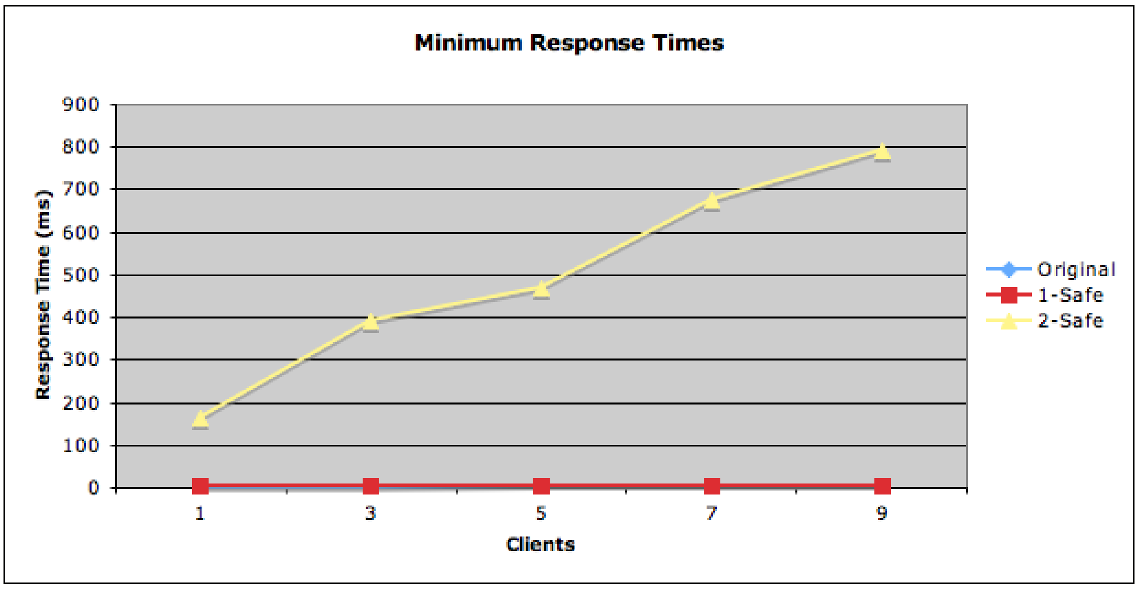


Figure 7.4: Minimum Response Times

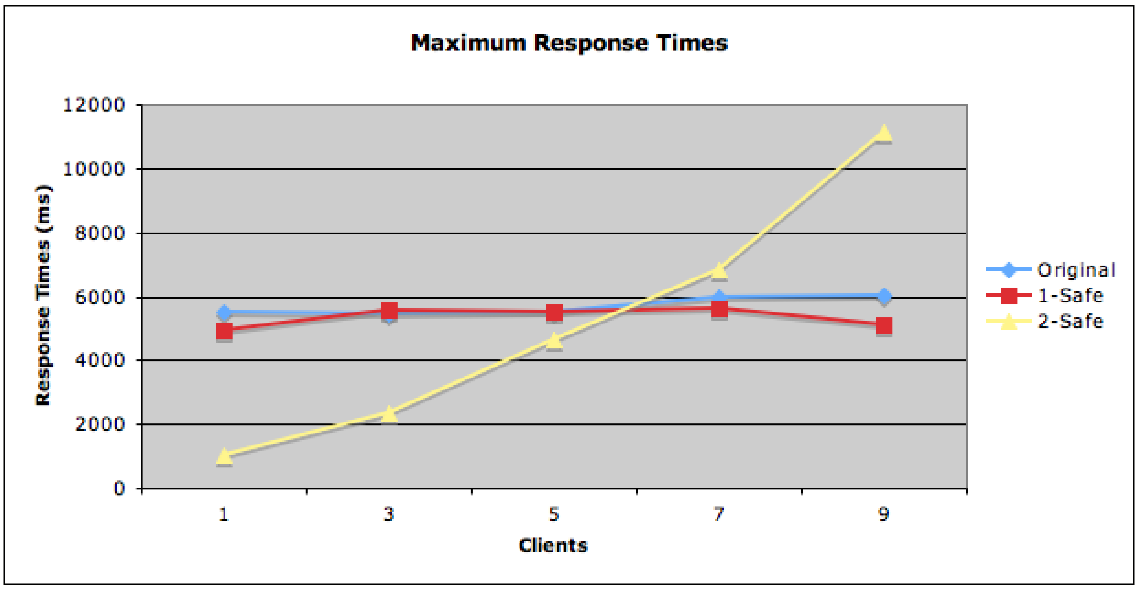


Figure 7.5: Maximum Response Times

7.3.3 Hardware Utilization

Another interesting aspect of the performance besides throughput and response times is the hardware utilization and the stress the Derby implementations impose on the server hardware. Measurements were taken every five seconds and the results can be seen in Figure 7.6 through 7.13. The results of these measurements accompany the results for the response times and throughput well.

The CPU utilization is a little higher for the one-safe approach than the original system, but the graphs are very similar with few exceptions. The two-safe approach however consumes more CPU resources. The original Derby has a lower disk utilization than both the one-safe and two-safe approach. The original derby has fewer resident transactions waiting at any time and therefore achieves higher throughput, lower response times, lower CPU utilization and also lower disk utilization. The higher disk utilization in both the one-safe and two-safe approaches can be explained by the logical log being written to disk every time it is shipped from the primary node. The log is shipped a lot more often when using two-safe replication and therefore the disk utilization is very high even if the throughput is low and response times high. The one-safe approach follows the same pattern as the original.

It can easily be seen by the results that most processing time is spent by waiting transactions, while few other resources are used: They all have to wait their turn and no transaction is allowed to commit until their log records are received at the log receiver. The measured hardware utilizations of the hot standby can be seen in Figure 7.7 and Figure 7.9. There are no surprises there as the one-safe has a higher throughput and therefore higher disk and cpu utilization even though the two-safe approach uses more system resources pr transaction.

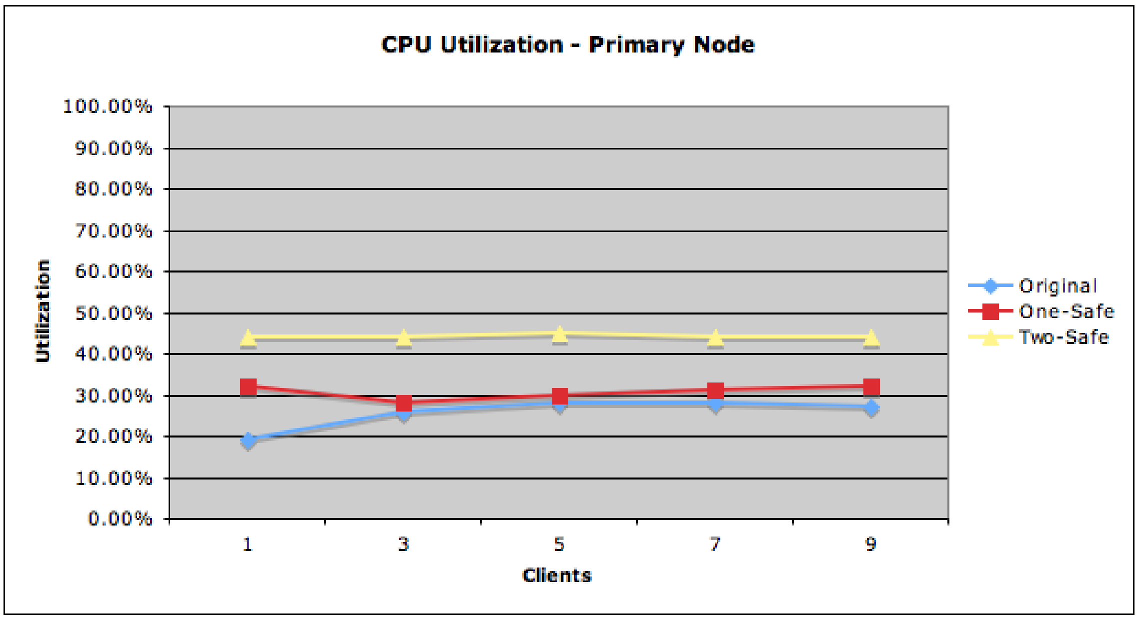


Figure 7.6: CPU Utilization - Primary Node

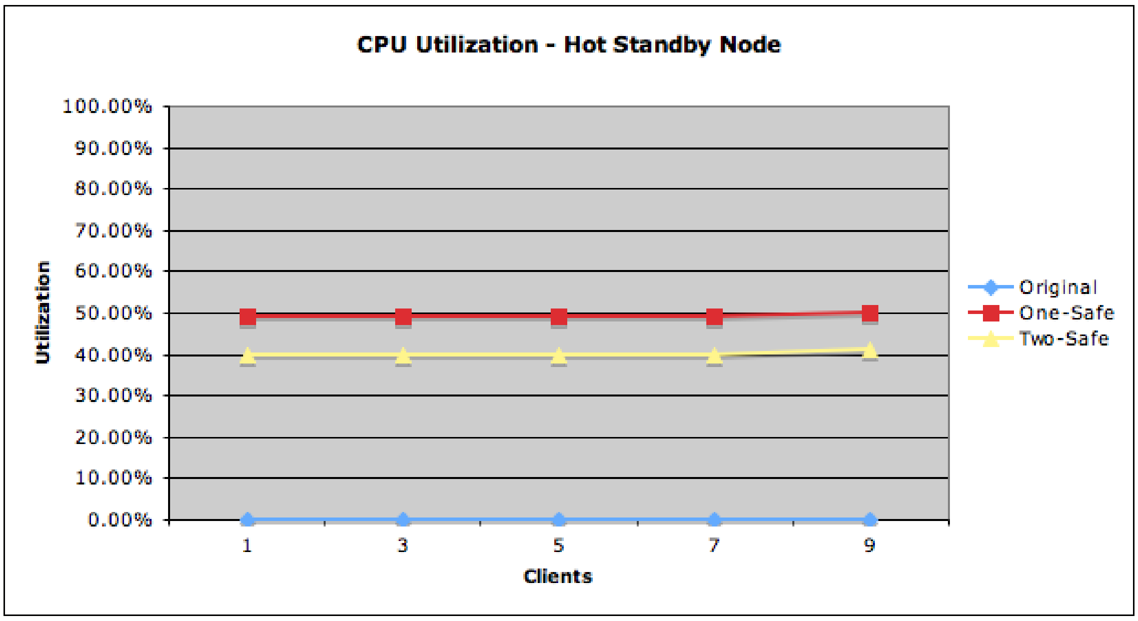


Figure 7.7: CPU Utilization - Hot Standby Node

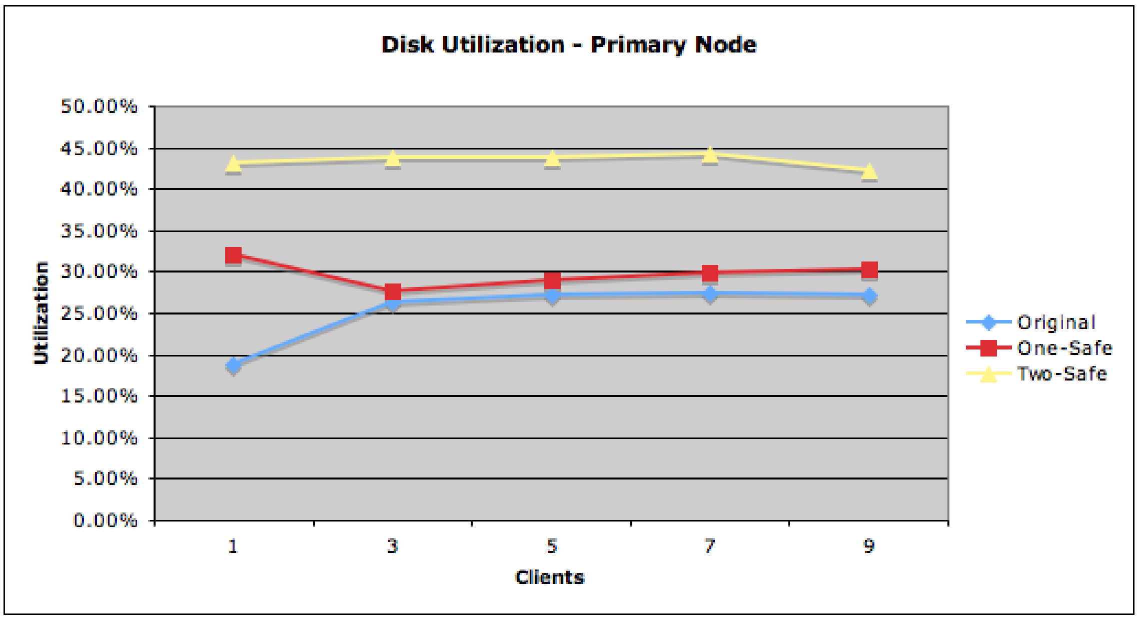


Figure 7.8: Disk Utilization - Primary Node

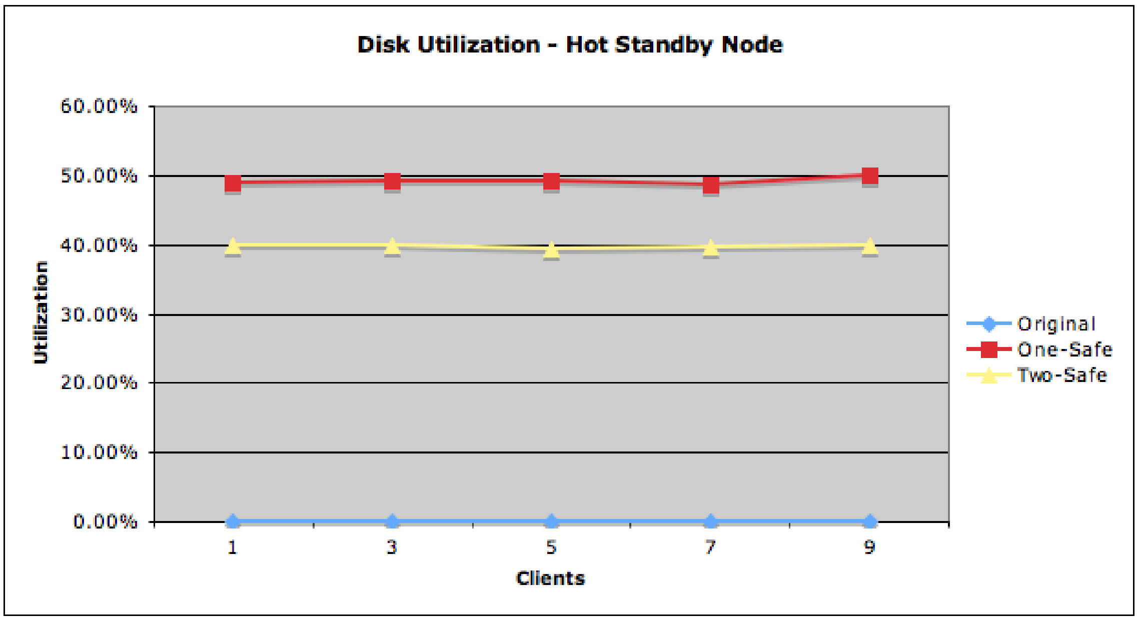


Figure 7.9: Disk Utilization - Hot Standby Node

Surprisingly however, the original Derby has a higher network utilization than the hot standby schemes. Due to a higher throughput and therefore a higher number of completed transactions, the system has a more intense conversation with the clients, requests and responses are more rapid which in turn consumes network utilization. If the test system had two separate lines it would be possible to view the client load and the load created by the replication. Due to the shared connection the results from the network utilization need to be normalized to see the additional strain the replication imposes on the system. This normalized average network utilization can be seen in Figure 7.12 and Figure 7.13 and shows the average network utilization per transaction.

As expected the two-safe approach has a higher utilization for each transaction as the number of logical log records shipped in each shipment are smaller and shipped more often which in turn generates a larger communication overhead. The one-safe implementation however ships the logical logs whenever they have exceeded a certain size. In the implemented benchmark every transaction consists of three update operations, one insert operation and a commit or abort operation, a total of five log records. In the two-safe implementation 2PC messages are first handed between the primary and hot standby and the log is shipped whenever a transaction commits. This in turn restricts the size of the log shipped greatly, while in one-safe more log records are sent at once, thus increasing network efficiency and reducing the network utilization.

If the amount of logs shipped each time is too great the hot standby will have trouble redoing them in time and transactions will pile up on the primary while waiting for the next shipment. If the buffer is too small however, the log will be shipped more often, depleting valuable system resources. By trial and error it is found that a buffer size of 250 log records is close to optimal regarding average throughput and response times in the one-safe approach. The buffer size was therefore set to 250 during these benchmarks.

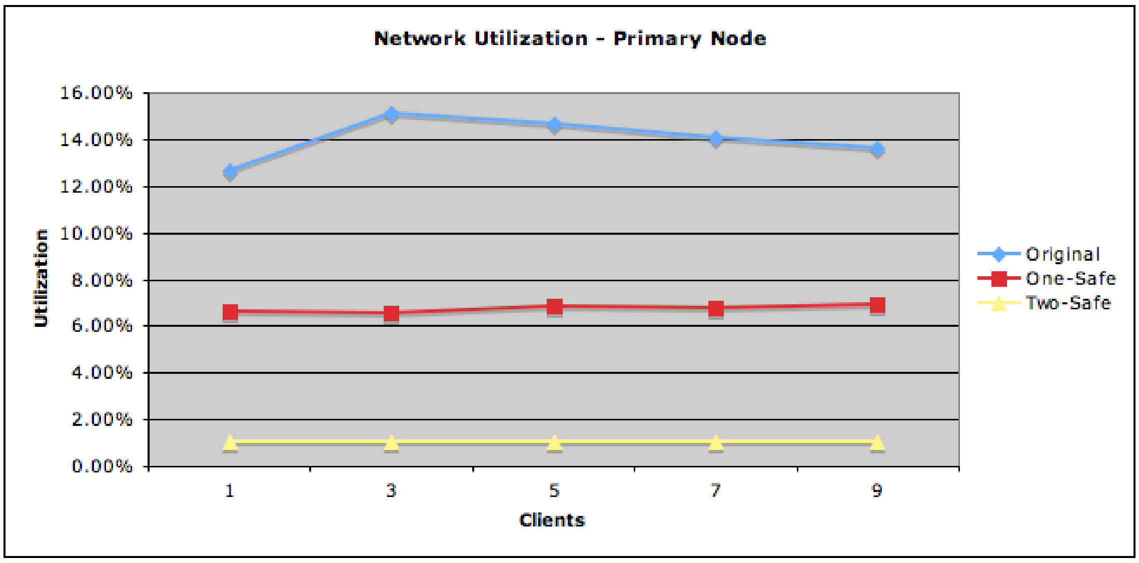


Figure 7.10: Network Utilization - Primary Node

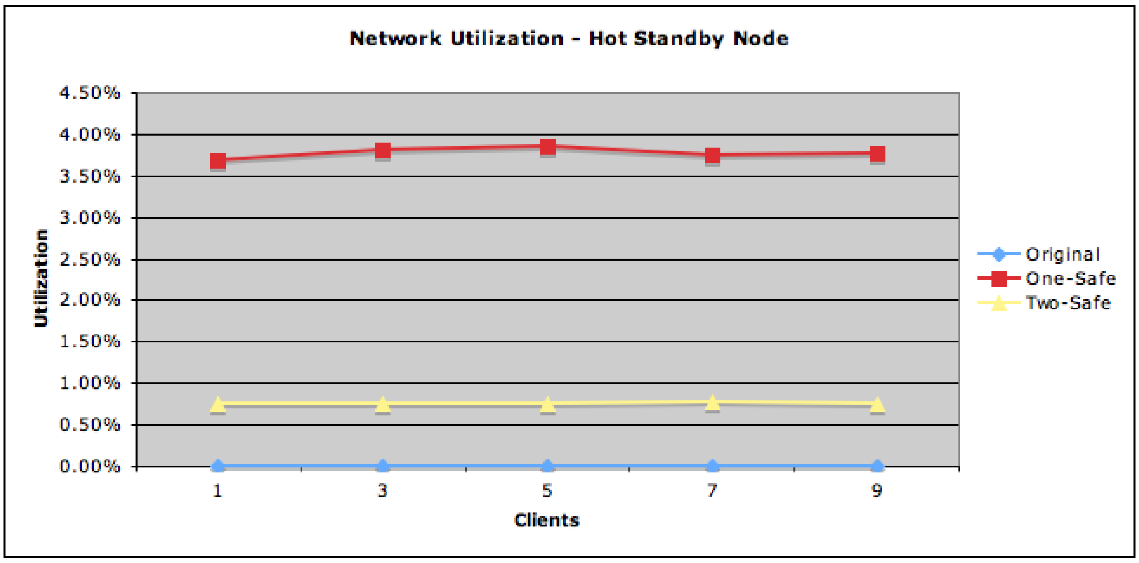


Figure 7.11: Network Utilization - Hot Standby Node

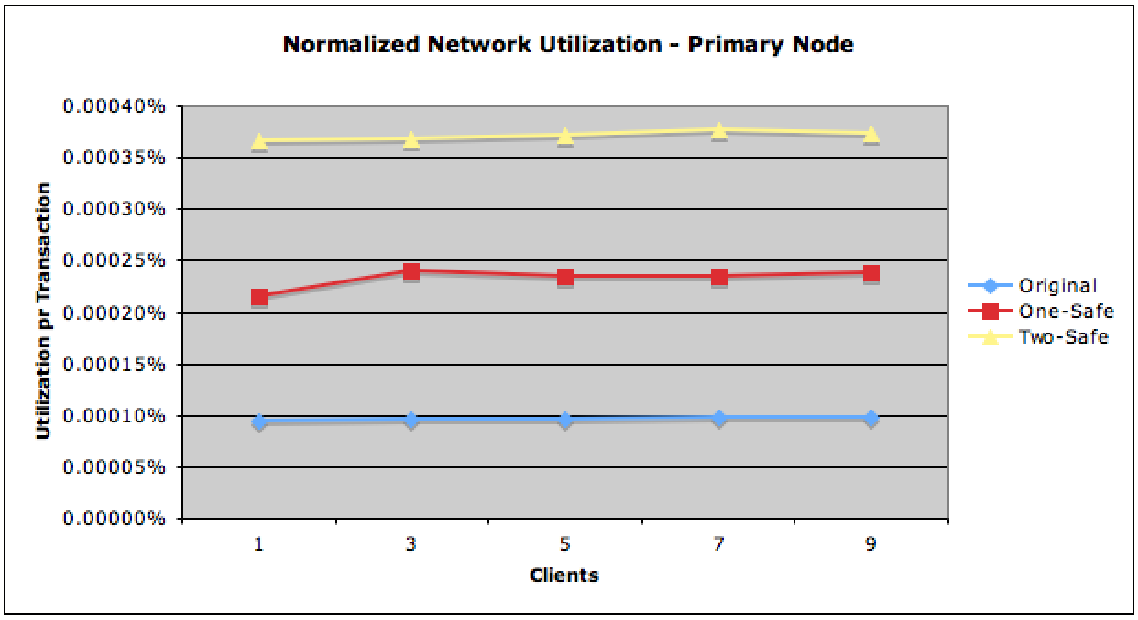


Figure 7.12: Normalized Network Utilization - Primary Node

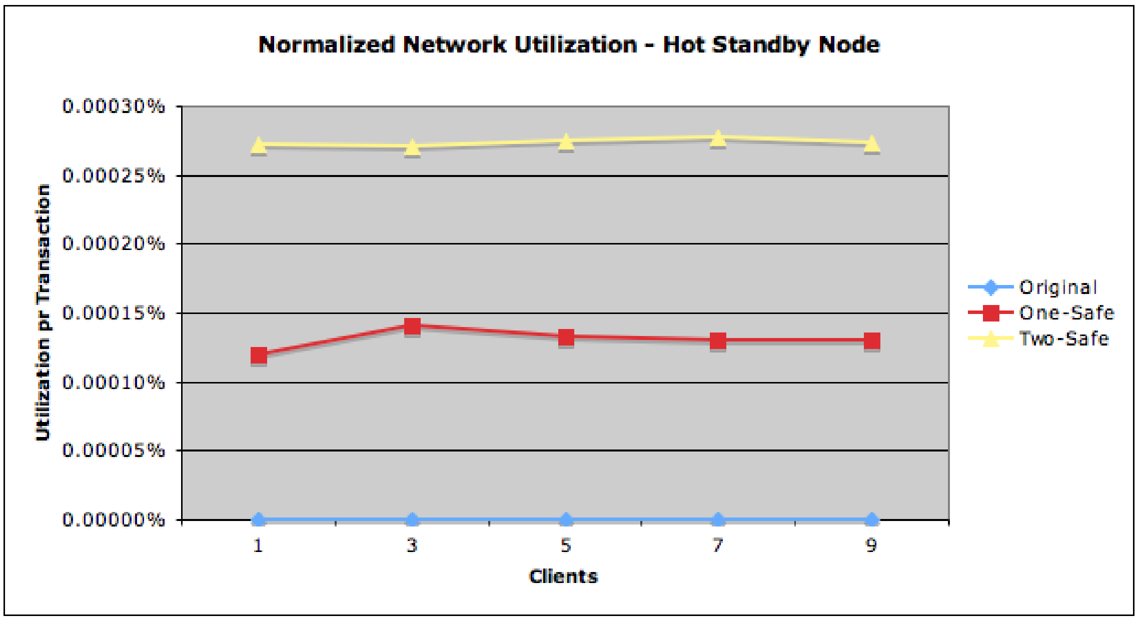


Figure 7.13: Normalized Network Utilization - Hot Standby Node

As can be seen by the hardware utilization figures it is quite clear that the bottleneck is not the communication. The main bottleneck is the waiting time on the hot standby database which is caused by queued transactions which in turn creates higher response times and lower overall throughput acchieved by the system. Great performance improvements could be achieved by improving the efficiency of the hot standby redo phase.

7.4 TPCB Consistency Check

After the TPC-B like benchmarks were completed a consistency check was performed to ensure that the hot standby and the primary is behaving correctly and that they both contain the same data after execution. The test goes through all the tables used in the TPC-B Benchmark, checking the size of the tables first, then checking the tuples from both databases and comparing them. If any two tuples are not equivalent the test is failed and stopped. The result of this test is cited below and the source code for this consistency check is added in Appendix A

```
[BRANCHES]
Checking size:
[OK] 1:1
Checking contents ...
[OK]
[/BRANCHES]

[TELLERS]
Checking size:
[OK] 10:10
Checking contents ...
[OK]
[/TELLERS]

[ACCOUNTS]
Checking size:
[OK] 100000:100000
Checking contents ...
......................................
........................................
........................................
........................................
........................................
[OK]
[/ACCOUNTS]

[HISTORY]
Checking size:
[OK] 24399:24399
```

```
Checking contents ...
.....................................
.........
[OK]
[/HISTORY]
```

Chapter 8

Conclusion

A lot of work was done both to learn the principles of log and database replication and to understand the inner workings of the Apache Derby DBMS, both the embedded server and the network server.

8.1 Derby: Replication and Availability

As shown, a fully functional prototype of the Apache Derby hot standby scheme has been created using logical logs, fail-fast takeovers and logical catchups after an internal up-to-crash recovery and reconnection. The performance of the system has been measured and even if the performance of the system is found to be worse than the original system, it is encouraging as there are still some room for great performance enhancements on the hot standby node. A correctness test has also been performed showing that the implemented DBMS as a whole[1] is working as it should, ensuring that both the databases are equivalent after each TPC-B Benchmark run. It has also been tested after multiple aborts, takeovers and catchups and no inconsistencies between the two databases have yet been found.

8.2 Performance Results

As shown, the hot standby replication scheme has a performance penalty, especially when it comes to throughput. The reason for this is simply that the hot standby cannot keep up with the primary database when the transaction load is high. When the system is run without the load on the hot standby[2] the performance is very close to the original network server. Also, when the benchmark is run on a slower network than the one used in the benchmark, the load is not as high and therefore the hot standby has no trouble keeping the pace of the primary. In these situations the hot standby scheme and the original network server run with the same throughput and response times. Hardware utilization is not surprisingly higher on the one-safe and two-safe replication schemes, but

[1]The primary and Hot Standby databases combined

[2]The received logs are not redone or flushed to disk

not so high that it has serious impact on the server running the database or regarding throughput or response times. The network interfaces, connections and shipment algorithms have been greatly improved from [4] and no longer poses any significant performance penalties.

8.3 Further Work

The prototype built has some further work to be done, the most important of these are listed here with some implementation suggestions.

8.3.1 Improved Redo Processing on Hot Standby

As the redo and undo processing on the hot standby is the main reason for the performance penalties is it preferable to improve how these updates are being done. In this project, received log records are converted to normal sql queries and run as normal sql statements on the embedded database. By somehow compiling prepared statements from these log records, the performance should be improved a lot. For even better performance the operations could be inserted directly into the database without going through the jdbc layer.

8.3.2 Slow Down on Catchup When Needed

When the database is under a lot of stress from high transaction loads the hot standby have problems getting caught up with the primary. The reason for this is that the primary creates log records faster than the hot standby can process them and thereby filling up the memory on the primary database until it crashes. Incoming connections should somehow be slowed down while the hot standby is catching up, much in the same way that they are set on wait when the hot standby have problems keeping up during normal processing.

8.3.3 Network Server Router

When the primary goes down an exception is thrown to the client. The client itself is then responsible for knowing where the hot standby is located and connecting to it to continue execution of the client software. However, if a dummy network server is created as a router for these jdbc connections the client no longer needs to know if it is connected to a database with replication or not. The Network Server Router knows the URL for the primary and hot standby databases and just forwards the jdbc calls to the correct database. On a primary database failure the router can then just pause the transaction until the takeover is completed and then forward the operations to the new primary. The router should be implemented as a process pair as well to ensure that it does not become a single-point-of-failure.

8.3.4 Complete 2PC Support for Two-Safe Replication

The two-safe replication scheme created in this project is very simplified in that it only simulates the 2PC messages handed between the primary and hot standby. By completing this protocol the hot standby checks if the operation is ok and then votes accordingly instead of just replying *VOTE_YES* every time

8.3.5 Failure Handling on Logical Log Write

If the database somehow crashes while writing the logical log to the disk, the logical log on disk becomes corrupted and it may be unreadable when the database starts up again. Instead of using a single-write mechanism, the logical log should be written with some measures of backup. Either that the old written log is copied first and then overwritten, so that if the logical log is unreadable, the older backup is read instead. As all the operations logged are idempotent[3] it does not matter if the operation is redone one time too many on the next catchup.

8.3.6 Support for Other SQL Statements

Support for more SQL statements should be added to the replication scheme to give the hot standby scheme the same functionality as the original network server. At the moment only update, insert and delete operations are available, while operations like drop, create and alter have not yet been implemented.

[3]They always yield the same result no matter how many times they are run

Chapter 9

Related Work

Some work that can be related to this project have been done, important and close resembling examples are ClustRa, Continental Pronto and Tandem RDF.

9.1 ClustRa

As shown earlier, ClustRa uses neighbor write-ahead-logging to log record operations. It is a main-memory logging technique that enables the system to commit operations without writing log records to disk. The log records are shipped to two failure-independent nodes and the log records are redone when they arrive at these nodes. The log records are primary key based and logical in the same manner as the logical log created by this project. The idea of making the Derby log receiver a Hot-standby is closely related to the replication approach used in ClustRa.

9.2 Continental Pronto

The Continental Pronto[10] is an algorithm created to combine the notions of local area replication and wide area replication. Instead of the low-level log shipping used in most systems it employs a higher level transaction shipping through an underlying communication abstraction, called HABcast. Continental Pronto, when used in local area replication acts like normal local area replication schemes. When used in wide area replications it is configureable to use both one-safe and two-safe replication.

9.3 Tandem RDF

Tandems Remote Duplicate Data Facility (RDF) is a service added to the Tandem Guardian 90 system[3] to mask environmental, operations and software faults by automatic replication to a second site. A process-pair, the *extractor*, is designed to continually inspect the log of the primary site and from time to time ship the log to another process-pair on the backup, the *receiver*, where the updates are reflected. The extractor sends *I'm alive* messages to the receiver while there is no activity. In this way the receiver will know if and when the primary server crashes. This approach is closely related to the one-safe replication done in this project. RDF also has the possibility for two-safe replication as well.

Bibliography

[1] Hector Garcia-Molina (2002): *Database Systems The Complete Book*, New Jersey: Prentice Hall

[2] Svein-Olaf Hvasshovd (1999): *Recovery in Parallel Database Systems 2nd Edition*, Gttingen: Friedr. Vieweg & Sohn Verlagsgesellschaft

[3] Jim Gray and Andreas Reuter (1993): *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann

[4] Egil Sørensen (2006): *Derby: Log to Neighbor Mode*

[5] Kexiang Hu, Sharad Mehrota and Simon Kaplan (1997): *An Optimized Two Safe Approach to Maintaining Remote Backup Systems*

[6] Kexiang Hu, Sharad Mehrota and Simon Kaplan (1998): *Failure Handling in an Optimized Two-Safe Approach to Maintaining Remote Backup Systems*

[7] Franklin, Michael J. et al (1992): *Crash Recovery in Client-Server EXODUS*

[8] Mohan, C. et al (1992): *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging*

[9] Mohan, C. (1986): *Transaction Management in the R* Distributed Database Management System*

[10] Svend Frølund and Fernando Pedone (2000): *Continental Pronto*

[11] Rune Humborstad, Maitrayi Sabaratnam, Svein-Olaf Hvasshovd, et al. (1997): *1-Safe Algorithms for Symmetric Site Configurations*

[12] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, et al. (1995): *The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response*

[13] Svein Erik Bratsberg, Svein-Olaf Hvasshovd, Øystein Torbjørnsen (1997): *Location and Replication Independent Recovery in a Highly Available Database*

[14] Transaction Processing Performance Council (1994): *TPC Benchmark Standard Specification Revision 2.0*

[15] The Apache DB Project: *Derby* (http://db.apache.org/derby/)

[16] Sun Microsystems: *Introduction to Apache Derby: TS-3154, 2006* (http://developers.sun.com/learning/javaoneonline/2006/coreenterprise/TS-3154.html)

Appendix A

The TPC-B-like Benchmark

A.1 The TPC-B Database Structure

The Entity Relations for the TPC-B database used in the benchmarking of Derby: Log to Neighbor is shown in Figure A.1. The simulated bank consists of one branch with 10 tellers and 100.000 accounts. All updates of the teller and account records are logged in the history table.

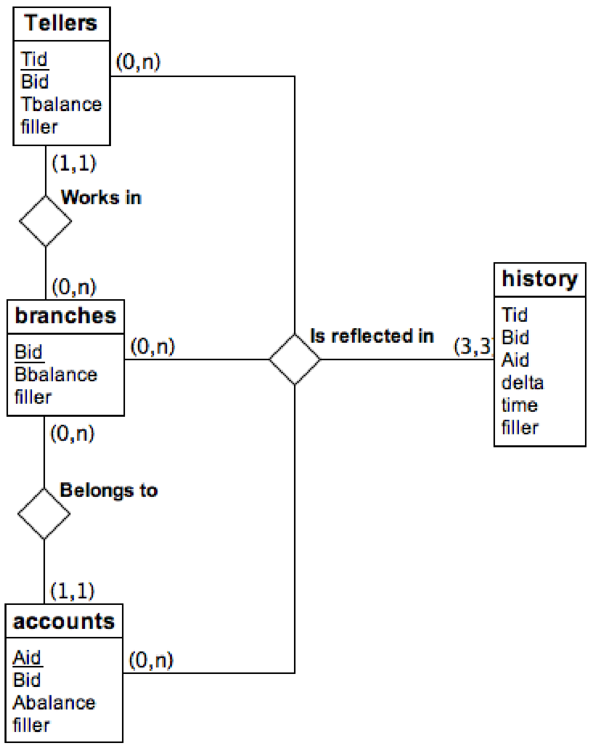


Figure A.1: Entity Relations for the TPC-B Database

A.2 Source Code

The java source code written to initialize the TPC-B like benchmark is shown in the following section. TPCBInit is responsible for creating the database, creating the tables and populate the required data. TPCBOperation is the implementation of a single TPC-B operation defined in [14] and finally TPCBenchmark is the class responsible for starting the TPC-B clients and doing the operations, throughput and response times are also collected here.

A.2.1 TPCBInit.java

```
package test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

public class TPCBInit {

public final static long tps = 1;
public final static long nbranches = 1;
public final static long ntellers = 10;
public final static long naccounts = 100000;
public final static long nhistory = 864000;

public static void main(String[] args){
System.out.println("[CONNECT]␣Connecting...");
try {
Class.forName("org.apache.derby.jdbc.ClientDriver").
    newInstance();
} catch (InstantiationException e) {
e.printStackTrace();
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
e.printStackTrace();
}

Connection conn = null;
Properties props = new Properties();
props.put("user", "user1");
props.put("password", "user1");

```

```
/*
The connection specifies create=true to cause
the database to be created. To remove the database,
remove the directory derbyDB and its contents.
The directory derbyDB will be created under
the directory that the system property
derby.system.home points to, or the current
directory if derby.system.home is not set.
*/

try{
conn = DriverManager.getConnection(
"jdbc:derby://prosjekt.xn--egilsrensen-kgb.dk:1527/TPCB;create=true",
props);

conn.setAutoCommit(false);

/*
Creating a statement lets us issue commands against
the connection.
*/
Statement s = conn.createStatement();



/*
* Create the tables
*/

System.out.println("[CREATE]␣Tables");
s.execute("CREATE␣TABLE␣branches␣(Bid␣NUMERIC(9),␣PRIMARY␣KEY(Bid),"+
"Bbalance␣NUMERIC(10),␣filler␣CHAR(88)" +
"␣DEFAULT␣'SYSTEM')");

s.execute("CREATE␣TABLE␣tellers␣("+
"Tid␣NUMERIC(9)␣PRIMARY␣KEY,"+
"Bid␣NUMERIC(9),"+
"Tbalance␣NUMERIC(10),"+
"filler␣CHAR(84)␣DEFAULT␣'SYSTEM'," +
"FOREIGN␣KEY␣(Bid)␣REFERENCES␣branches(Bid))");

s.execute("CREATE␣TABLE␣accounts␣("+
"Aid␣NUMERIC(9)␣PRIMARY␣KEY,"+
"Bid NUMERIC(9),"+
"Abalance NUMERIC(10),"+
"filler CHAR(84) DEFAULT 'SYSTEM'," +
"FOREIGN KEY (Bid) REFERENCES branches(Bid)"+
")");

s.execute("CREATE TABLE history ("+
"Tid NUMERIC(9) REFERENCES tellers(Tid),"+
"Bid NUMERIC(9) REFERENCES branches(Bid),"+
"Aid NUMERIC(9) REFERENCES accounts(Aid),"+
"delta NUMERIC(10),"+
"time TIMESTAMP,"+
"filler CHAR(22) DEFAULT 'SYSTEM'" +
")");

s.close();

/*
* Prime the database
*/

PreparedStatement ps;

int affRows = 0;
System.out.println("[INSERT] branches");
ps = conn.prepareStatement("INSERT INTO branches(Bid, 
    Bbalance) VALUES (?,0)");
for (int i = 1; i <= nbranches*tps; i++){
ps.setInt(1, i);
affRows += ps.executeUpdate();
}

System.out.println("[INSERT] tellers");
ps = conn.prepareStatement("INSERT INTO tellers(Tid,Bid,
    Tbalance) VALUES (?,?,0)");
for (int i = 1; i <= ntellers*tps; i++){
ps.setInt(1, i);
ps.setLong(2, 1);
affRows += ps.executeUpdate();
}

System.out.println("[INSERT] accounts");
ps = conn.prepareStatement("INSERT INTO accounts(Aid,Bid,
    Abalance) VALUES (?,?,0)");
for (int i = 1; i <= naccounts*tps; i++){
```

```
ps.setInt(1, i);
ps.setLong(2, 1);
affRows += ps.executeUpdate();
}

System.out.println("Succesfully set up and primed the 
    database");
System.out.println("Affected rows: " + affRows);
System.out.println("[DONE]");

ps.close();

conn.commit();

conn.close();

} catch (SQLException e){
e.printStackTrace();
}


}
}
```

A.2.2 TPCBOperation.java

```

package test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.Properties;

public class TPCBOperation {
private Connection conn;
private Properties props;
private PreparedStatement ps1;
private PreparedStatement ps2;
private PreparedStatement ps3;
private PreparedStatement ps4;
private PreparedStatement ps5;


public TPCBOperation(){
try {
Class.forName("org.apache.derby.jdbc.ClientDriver").
    newInstance();

} catch (InstantiationException e) {
e.printStackTrace();
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
e.printStackTrace();
}

conn = null;
props = new Properties();
props.put("user", "user1");
props.put("password", "user1");

try{

conn = DriverManager.getConnection(
"jdbc:derby://prosjekt.xn--egilsrensen-kgb.dk:1527/TPCB;
    create=true",
props);

conn.setAutoCommit(false);

ps1 = conn.prepareStatement("UPDATE accounts SET Abalance 
    = Abalance + ? WHERE Aid = ?");
ps2 = conn.prepareStatement("SELECT Abalance FROM accounts
     WHERE Aid = ?");
ps3 = conn.prepareStatement("UPDATE tellers SET Tbalance =
     Tbalance + ? WHERE Tid = ?");
ps4 = conn.prepareStatement("UPDATE branches SET Bbalance 
    = Bbalance + ? WHERE Bid = ?");
ps5 = conn.prepareStatement("INSERT INTO history(Tid, Bid,
     Aid, delta, time) VALUES(?,?,?,?,?)");


} catch (SQLException e){
e.printStackTrace();
}

}




public synchronized long doOneOperation(long bid, long tid
    , long aid, long delta){
try{

long abalance = -1;


ps1.setLong(1, delta);
ps1.setLong(2, aid);
ps1.executeUpdate();

ps2.setLong(1, aid);
ResultSet rs = ps2.executeQuery();

if (rs.next())
abalance = rs.getLong("Abalance");

ps3.setLong(1, delta);
ps3.setLong(2, tid);
ps3.executeUpdate();
```

```

ps4.setLong(1, delta);
ps4.setLong(2, bid);
ps4.executeUpdate();

ps5.setLong(1, tid);
ps5.setLong(2, bid);
ps5.setLong(3, aid);
ps5.setLong(4, delta);
Timestamp time = new Timestamp(System.currentTimeMillis())
    ;
ps5.setTimestamp(5, time);
ps5.execute();


conn.commit();

return abalance;

} catch (SQLException e){
e.printStackTrace();
return -1;
}

}




}
```

A.2.3 TPCBenchmark.java

```java
package test;

public class TPCBBenchmark {


public static void main(String[] args){


TPCBBenchmark benchmark = new TPCBBenchmark();

// Number of concurrent clients
int numClients = 9;
benchmark.startBenchMark(numClients);

}

public void startBenchMark(int numClients){
System.out.println("Starting␣benchmark␣with␣" + numClients
    + "␣clients");
for (int i = 1; i <= numClients; i++){
new Thread(new TPCBClient(i)).start();
}

}

private class TPCBClient extends Thread{

private int clientId;

public TPCBClient(int clientId){
this.clientId = clientId;
}



public void run(){
// ten minutes
System.out.println("Running␣client␣" + clientId);
long runTime = 1000 * 60 * 15;
long start = System.currentTimeMillis();
long end = start + runTime;
long totalResponseTime = 0;
int completedTransactions = 0;
long maxResponseTime = -1;
long minResponseTime = -1;
TPCBOperation operation = new TPCBOperation();

while (System.currentTimeMillis() <= end){
long tid = Math.round(Math.random() * (TPCBInit.ntellers*
    TPCBInit.tps - 1)) + 1;
long aid = Math.round(Math.random() * (TPCBInit.naccounts*
    TPCBInit.tps - 1)) + 1;

long delta = Math.round(Math.random() * 1000);

long transStartTime = System.currentTimeMillis();
operation.doOneOperation(1, tid, aid, delta);
long responseTime = System.currentTimeMillis() -
    transStartTime;

if (maxResponseTime < responseTime || maxResponseTime ==
    -1)
maxResponseTime = responseTime;
if (minResponseTime > responseTime || minResponseTime ==
    -1)
minResponseTime = responseTime;

totalResponseTime += responseTime;
++completedTransactions;

}

System.out.println("[" + clientId + "] Completed 
    transactions: " + completedTransactions);
System.out.println("[" + clientId + "] Transactions per 
    second: " + (double) Math.round(((double)
    completedTransactions / (runTime / 1000)) * 100) / 100)
    ;
System.out.println("[" + clientId + "] Average response 
    time: " + totalResponseTime / completedTransactions);
System.out.println("[" + clientId + "] Minimum response 
    time: " + minResponseTime);
System.out.println("[" + clientId + "] Maximum response 
    time: " + maxResponseTime);

}
}
}
```

A.2.4 TPCB Consistency Test

A consistency test is added to the TPCB benchmark to check if the contents of both the primary and the hot standby are the same.

```java


package test.correctness;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

/**
 *
 * The TPCBCorrectionCheck is a class responsible for
     checking the
 * consistency in both the primary and the hot standby
     database. Both
 * databases need to be started in the original server/
     client mode.
 * The contents of the TPCB databases are then compared to
      eachother.
 *
 * @author Egil Srensen
 *
 */

public class TPCBCorrectionCheck {
/**
 * Connection to the primary database
 */
private Connection primConn;
/**
 * Connection to the hot standby database
 */
private Connection hsbConn;
/**
 * Properties including username and password
 */
private Properties props;
/**
 * Statement used to execute sql queries to the primary
     database
 */
private Statement primS;
/**
 * Statement used to execute sql queries to the primary
     database
 */
private Statement hsbS;

/**
 * Main method to start the test
 *
 * @param args  No arguments are used
 */
public static void main(String[] args){
TPCBCorrectionCheck check = new TPCBCorrectionCheck();
check.checkConsistency();
}

/**
 * Initialize the test and set up connections
 *
 */
public TPCBCorrectionCheck(){
try {
Class.forName("org.apache.derby.jdbc.ClientDriver").
    newInstance();
} catch (InstantiationException e) {
e.printStackTrace();
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (ClassNotFoundException e) {
e.printStackTrace();
}
hsbConn = null;
primConn = null;
props = new Properties();
props.put("user", "user1");
props.put("password", "user1");
try{
String primUrl = "lachdanan.egilnett.com";
String hsbUrl = "kratas.egilnett.com";
primConn = DriverManager.getConnection("jdbc:derby://" +
    primUrl + ":1527/TPCB;create=true", props);

```

```
hsbConn = DriverManager.getConnection("jdbc:derby://" +
    hsbUrl + ":1527/TPCB;create=true", props);
} catch (SQLException e){
e.printStackTrace();
}

}



/**
 * Check the consistency of the two databases (primary and
     hot standby). The size
 * and contents of the tables BRANCHES, TELLERS, ACCOUNTS
    and HISTORY.
 *
 * If any irregularities are found, the error is shown and
     the check is stopped.
 *
 */
public void checkConsistency(){
try {
// Temporary variables
String query;
ResultSet primRs;
ResultSet hsbRs;
int primSize;
int hsbSize;
int diff;
boolean error;
int counter;

// Create the statements
primS = primConn.createStatement();
hsbS = hsbConn.createStatement();


// Check the BRANCHES-table
System.out.println("[BRANCHES]");
error = false;


// Getting size
query = "select␣count(bid)␣as␣numBranches␣from␣branches";
primRs = primS.executeQuery(query);
```

```
hsbRs = hsbS.executeQuery(query);

primRs.next();
primSize = primRs.getInt("numBranches");
hsbRs.next();
hsbSize = hsbRs.getInt("numBranches");

primRs.close();
hsbRs.close();

System.out.println("Checking size:");
if (primSize == hsbSize)
System.out.print("[OK] ");
else{
System.out.print("[ERROR] ");
error = true;
}
System.out.print(primSize+":"+hsbSize+"\n");

// Quit on error
if (error)
System.exit(-1);


// Getting contents
query = "select bid, bbalance, filler from branches order
    by bid asc";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);
System.out.println("Checking contents ...");

diff = 0;
counter = 0;
while (primRs.next() && hsbRs.next()){
if ((primRs.getInt("bid") != hsbRs.getInt("bid")) || (
    primRs.getInt("bbalance") != hsbRs.getInt("bbalance")))
    {
System.out.print("\n");
diff++;
System.out.println("[ERROR] bid: " + primRs.getInt("bid")+
    ":"+hsbRs.getInt("bid")+", bbalance: " + primRs.getInt(
    "bbalance") + ":" + hsbRs.getInt("bbalance"));
}
if (counter % 20000 == 0 && counter != 0)
System.out.print("\n");
```

```
if (counter % 500 == 0 && counter != 0)
System.out.print(".");
counter++;
}
System.out.print("\n");
primRs.close();
hsbRs.close();


if (diff == 0){
System.out.println("[OK]");
}
else{
System.out.println("[ERROR]␣" + diff + "␣row(s)␣do(es)␣not
    ␣match␣in␣the␣two␣databases.␣See␣above");
System.exit(−1);
}

// Finished checking the BRANCHES table
System.out.println("[/BRANCHES]");
System.out.println("");
error = false;

// Check the TELLERS table
System.out.println("[TELLERS]");


// Getting size
query = "select␣count(tid)␣as␣numTellers␣from␣tellers";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);

primRs.next();
primSize = primRs.getInt("numTellers");
hsbRs.next();
hsbSize = hsbRs.getInt("numTellers");
primRs.close();
hsbRs.close();

System.out.println("Checking␣size:");
if (primSize == hsbSize)
System.out.print("[OK]␣");
else{
System.out.print("[ERROR]␣");
error = true;
```

```
}

System.out.print(primSize+":"+hsbSize+"\n");

// Quit on error
if (error)
System.exit(-1);

// Getting contents
query = "select tid, bid, tbalance, filler from tellers order by tid asc";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);
System.out.println("Checking contents ...");

diff = 0;
counter = 0;
while (primRs.next() && hsbRs.next()){
if ((primRs.getInt("tid") != hsbRs.getInt("tid")) || (primRs.getInt("bid") != hsbRs.getInt("bid")) || (primRs.getInt("tbalance") != hsbRs.getInt("tbalance"))){
diff++;
System.out.print("\n");
System.out.println("[ERROR] tid: " + primRs.getInt("tid")+":"+hsbRs.getInt("tid")+", bid: " + primRs.getInt("bid")+":"+hsbRs.getInt("bid")+", tbalance: " + primRs.getInt("tbalance") + ":" + hsbRs.getInt("tbalance"));
}
if (counter % 20000 == 0 && counter != 0)
System.out.print("\n");
if (counter % 500 == 0 && counter != 0)
System.out.print(".");
counter++;
}
System.out.print("\n");
primRs.close();
hsbRs.close();

if (diff == 0){
System.out.println("[OK]");
}
else{
System.out.println("[ERROR] " + diff + " row(s) do(es) not match in the two databases. See above");
System.exit(-1);
```

```java
}

// Finished checking the TELLERS table
System.out.println("[/TELLERS]");
System.out.println("");
error = false;

// Checking the ACCOUNTS table
System.out.println("[ACCOUNTS]");

// Getting size
query = "select␣count(aid)␣as␣numAccounts␣from␣accounts";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);

primRs.next();
primSize = primRs.getInt("numAccounts");
hsbRs.next();
hsbSize = hsbRs.getInt("numAccounts");
primRs.close();
hsbRs.close();


System.out.println("Checking␣size:");
if (primSize == hsbSize)
System.out.print("[OK]␣");
else{
System.out.print("[ERROR]␣");
error = true;
}

System.out.print(primSize+":"+hsbSize+"\n");

// Quit on error
if (error)
System.exit(-1);

// Getting contents
query = "select␣aid,␣bid,␣abalance,␣filler␣from␣accounts␣
    order␣by␣aid␣asc";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);
System.out.println("Checking␣contents␣...");

diff = 0;
```

```
counter = 0;
while (primRs.next() && hsbRs.next()){
if ((primRs.getInt("aid") != hsbRs.getInt("aid")) || (
    primRs.getInt("bid") != hsbRs.getInt("bid")) || (primRs
    .getInt("abalance") != hsbRs.getInt("abalance"))){
System.out.print("\n");
diff++;
System.out.println("[ERROR]␣aid:␣" + primRs.getInt("aid")+
    ":"+hsbRs.getInt("aid")+",␣bid:␣" + primRs.getInt("bid"
    )+":"+hsbRs.getInt("bid")+",␣abalance:␣" + primRs.
    getInt("abalance") + ":" + hsbRs.getInt("abalance"));
}
if (counter % 20000 == 0 && counter != 0)
System.out.print("\n");
if (counter % 500 == 0 && counter != 0)
System.out.print(".");
counter++;
}
System.out.print("\n");
primRs.close();
hsbRs.close();

if (diff == 0){
System.out.println("[OK]");
}
else{
System.out.println("[ERROR]␣" + diff + "␣row(s)␣do(es)␣not
    ␣match␣in␣the␣two␣databases.␣See␣above");
System.exit(-1);
}

// Finished checking the ACCOUNTS table
System.out.println("[/ACCOUNTS]");
System.out.println("");
error = false;

// Check the HISTORY table
System.out.println("[HISTORY]");


// Getting size
query = "select␣count(tid)␣as␣numHistory␣from␣history";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);

```

```
primRs.next();
primSize = primRs.getInt("numHistory");
hsbRs.next();
hsbSize = hsbRs.getInt("numHistory");
primRs.close();
hsbRs.close();


System.out.println("Checking size:");
if (primSize == hsbSize)
System.out.print("[OK] ");
else{
System.out.print("[ERROR] ");
error = true;
}

System.out.print(primSize+":"+hsbSize+"\n");

// Quit on error
if (error)
System.exit(-1);

// Getting contents
query = "select aid, bid, tid, delta, time, filler from 
   history order by tid, bid, aid, time asc";
primRs = primS.executeQuery(query);
hsbRs = hsbS.executeQuery(query);
System.out.println("Checking contents ...");

diff = 0;
counter = 0;
while (primRs.next() && hsbRs.next()){
if ((primRs.getInt("tid") != hsbRs.getInt("tid")) || (
   primRs.getInt("bid") != hsbRs.getInt("bid")) || (primRs
   .getInt("aid") != hsbRs.getInt("aid")) || (primRs.
   getInt("delta") != hsbRs.getInt("delta")) || ! primRs.
   getTime("time").equals(hsbRs.getTime("time"))){
System.out.print("\n");
diff++;
System.out.println("[ERROR] aid: " + primRs.getInt("aid")+
   ":"+hsbRs.getInt("aid")+", tid: " + primRs.getInt("tid"
   )+":"+hsbRs.getInt("tid")+", bid: " + primRs.getInt("
   bid") + ":" + hsbRs.getInt("bid")+", delta: " + primRs.
   getInt("delta") + ":" + hsbRs.getInt("delta")+", time: 
   " + primRs.getTime("time") + ":" + hsbRs.getTime("time"
    ));
}
if (counter % 20000 == 0 && counter != 0)
System.out.print("\n");
if (counter % 500 == 0 && counter != 0)
System.out.print(".");
counter++;
}
System.out.print("\n");
primRs.close();
hsbRs.close();

if (diff == 0){
System.out.println("[OK]");
}
else{
System.out.println("[ERROR]␣" + diff + "␣row(s)␣do(es)␣not
    ␣match␣in␣the␣two␣databases.␣See␣above");
System.exit(-1);
}

// Finished checking the HISTORY table
System.out.println("[/HISTORY]");

// Close the resultsets
primRs.close();
hsbRs.close();

// Close statements and connections
primS.close();
hsbS.close();
primConn.close();
hsbConn.close();

} catch (SQLException e) {
e.printStackTrace();
}
}





}
```

Appendix B

Arguments for the Network Server

In addition to the standard arguments that can be added to derby on startup of the network server the following have been added for hot standby support:

- **hsb** A flag to indicate if the server should be started as primary or hot standby, can be either ”primary” or ”standby”
- **hsburl** The url to the hot standby, used if the database is started as primary

Examples:
To start the server as primary with a hotstandby located at hsb.example.com:
java -server org.apache.derby.drda.NetworkServerControl start -hsb primary -hsburl hsb.example.com

To start the corresponding server as hot standby:
java -server org.apache.derby.drda.NetworkServerControl start -hsb standby

These two arguments can of course be combined with all of the arguments already available in derby.

Appendix C

Source Code

Due to the large amount of source code created in this project it is supplied as a zip file to this report. The changed and created classes are listed below.

C.1 The Logical Log System

- org.apache.derby.iapi.store.logical.log.LogicalImage
- org.apache.derby.iapi.store.logical.log.LogicalLoggable
- org.apache.derby.impl.store.logical.log.LogicalLog
- org.apache.derby.impl.store.logical.log.LogicalLogger
- org.apache.derby.impl.store.logical.log.LogicalLogRecord
- org.apache.derby.impl.store.logical.opearations.LogicalAbortOperation
- org.apache.derby.impl.store.logical.opearations.LogicalCommitOperation
- org.apache.derby.impl.store.logical.opearations.LogicalDeleteOperation
- org.apache.derby.impl.store.logical.opearations.LogicalImage
- org.apache.derby.impl.store.logical.opearations.LogicalInsertOperation
- org.apache.derby.impl.store.logical.opearations.LogicalOperation
- org.apache.derby.impl.store.logical.opearations.LogicalOperationFactory
- org.apache.derby.impl.store.logical.opearations.LogicalUpdateOperation

C.2 The SQL Layer

- org.apache.impl.sql.execute.RowChangerImpl
- org.apache.impl.sql.conn.GenericLanguageConnectionContext

C.3 The Communications Service

- org.apache.derby.impl.services.net.LogicalServer
- org.apache.derby.impl.services.net.LogicalClient
- org.apache.derby.impl.services.net.NetworkPayload
- org.apache.derby.impl.services.net.receiver.LogReceiver

- org.apache.derby.impl.services.net.shipper.LogShipper
- org.apache.derby.impl.services.net.shipper.LogicalCatchUp

C.4 The Hot Standby Service

- org.apache.derby.impl.services.hotstandby.HotStandbyController
- org.apache.derby.impl.services.hotstandby.Database

C.5 The Network Server

- org.apache.derby.impl.drda.NetworkServerControlImpl
- org.apache.derby.impl.drda.HotStandbyPoller
- org.apache.derby.impl.drda.net.LogicalClient
- org.apache.derby.imp.drda.net.LogicalServer