Pre-Diploma Thesis

Extending A Data Replication Technique Using DataLink

Institute for Computer Science

UNIVERSITY OF ROSTOCK

Major: Databases

Author: Guido Rost

Tutor: Holger Meyer

Revision: June 27, 2000

Contents

List of Figures

List of Tables

1 Introduction

The Replication of data is a technique for achieving better performance and fault-tolerance in distributed database systems. It's idea is to access only local data or a fast accessible nearby copy. That gives the opportunity of continuous working in case of network failures or no network connection at all.

Database replication has become an increasingly exploited technology. It is an important technique for ensuring high availability. Commercial database vendors such as IBM and Sybase have supported replication for a number of years. But replication is by no means a "done deal". It is still an area of active research.

IBM has developed a new technology, called Data-Links, which extends the Relational Database Management System (RDBMS) to data stored in external operating system files. Data-Links provide the vital integration between the RDBMS and the file system through extensions to a database. This paper is about a prototype on how to replicate this new Data-Link datatype and its corresponding data in the file system using IBM's DB2 Universal Database (UDB).

In todays business, most of the data is still living in file systems and not in a formal DBMS environment. The sheer volume of data stored in files is much bigger than data in a DBMS. And it continuously grows in a much faster rate. This difference in growth rate increases with the advent of Internet/Web applications with its huge amount of multimedia data. To keep this data in the file system and not moving them in an RDBMS is useful and mostly necessary. These files are especially unstructured and semi-structured data such as images, documents, engineering drawings, email massages, video clips, presentations, spreadsheets, etc. They are very often related in some way to traditional data stored in the RDBMS.

These applications have good reasons for storing their data in the file system but need robust management capabilities without having to move those data into the DBMS. For example Internet organizations want to provide users with access to a variety of information in many different formats and engineering companies need their drawing files for their computer aided design (CAD/CAM) applications. All of these applications clearly illustrate the long-term challenge for an comprehensive content-management system that integrate external data and files with traditional business data. IBM's Data-Link technology is covering exactly these requirements.

The following chapter 2 gives an overview of the principles of data replication and will look at some replication methods more detailed. Afterwards, chapter 3 introduces IBMs replication product DPropR. First, it tries to classify the product by comparing the mechanisms it based on. Then the functioning of DpropR will be explained. In chapter 4, the feature Data Links of DB2UDB is introduced. In chapter 5, the problems the prototype comes with are explained and disscussed. The prototype itself will be presented and discussed in chapter 6. Chapter 7 gives an overview of the implementation of the prototype, then the perfomance will be disscussed by executing performance measurements. Chapter 8 concludes of what was done and what has to be done to make the prototype more usable in a real world scenario.

2 Overview of Replication

In the last years, distributed database systems became more important. This trend is recognizable even to the more classical database applications using client-server-applications instead of mainframes. Distributed databases are used in many new application areas as mobile computing, work-flow management, concurrent engineering or data warehousing too.
An assumption for better fault tolerance and performance in a distributed system is the replication of data. That ensures an undisturbed working environment even with network failures, and increases performance. On the other hand it increases the risk of data inconsistencies and efforts of update operations. In order to find an appropriate solution for this trade-off many methods for the management of replicated data have been proposed in the literature.

This chapter introduces different replication methods and surveys their principles. It also mentions advantages and problems of the presented methods.

The content of this chapter is based to a large extent on [1].

2.1 Ensuring Data Consistency

A very important requirement for replication methods is to ensure data consistency. To describe the correctness of replicated databases, the term of *One-Copy-Serializability* is mostly used. If a schedule is one-copy-serializable it keeps the replicated database in the same consistent state as it was before.

One-copy-serializability:
A replication method is one-copy-serializable if the transaction processed over a replicated object residing in the same initial state lead to the same object value as if these transaction were applied to a non-replicated object.

That means that there is one definite value assigned to a replicated object at any time. A replication method which complies with the requirement of One-Copy-Serializability must ensure the following points:

1. There can be only one successful write operation at a given time on an replicated object.

2. Two read operations executed from any computers at any time determine the same object value of the replicated object. And that object values represent the update of the last successful write operation.

The requirement is realizable in two ways:

- syntactic methods
  Data integrity at syntactic methods depends only on the order of the accessing transactions, the order of read – write operations.

- semantic methods
  Semantic methods make use of the semantics of transactions, for instance the possibility of changing the processing order of two operations (commutativity). The correctness of some semantic approaches can only be defined using integrity conditions of the database. Other approaches use special knowledge of the database.
  Semantic methods are basically not as general applicable as syntactic methods because syntactic methods do not comply with the semantic premises that are needed. On the other hand semantic methods are very attractive because of their semantic knowledge, it can help to reduce communication and synchronization work and that means network traffic. The semantic knowledge can replace the synchronization over all the copies. In best case, it could be possible that only the local copy needs to be updated because the replication method has semantic knowledge of the current update. On the other hand the number of copies which to update asynchronously can get very high and it can be very complicated up to not possible to read a consistent database value.

In replicated databases, you must define the amount of copies which to update synchronously with the commit of a transaction and which to update asynchronously. There must be at least one copy which will be updated synchronously. The more copies you update synchronously the more copies will be in a consistent state. On the other hand that means that there is more work to apply before a transaction commits and the possibility of failures that can occur before the commit is higher too. For instance if one copy is not reachable and therefore it can not be updated synchronously. In this case the transaction commit will be blocked. Asynchronous updates don't have that problems and because of that they don't block a transaction commit.
There are several copy-update methods that try to solve the problem that synchronous replication comes with. To have always the necessary replicas

available is a strong requirement for a transaction commit. But that's not always the case.
Depending on the update method the amount of copies which have to be updated synchronously varies. It can be only one copy updated synchronously for absolutistic methods (introduced below) up to as most as possible (all if possible) in *Read-One-based* methods.

In the following the most common syntactic copy-update methods are introduced and how these methods meet the requirements above is discussed.

2.2 Copy-Update Methods

2.2.1 Voting

Voting is a democratic approach. The synchronization of accesses to each copies will be done by voting. Every copy gets a certain amount of votes, mostly one. A transaction is allowed to access an object if there is a quorum of copies which agrees to access the object.
There are often used different quorums, quorums for read operations Q_R and quorums for write operations Q_W. If the following overlapping rules are complied with for choosing read and write quorums data inconsistencies cannot occur anymore(see 2.1 on page 3):

- write/write - overlapping rule:
  $2 * Q_W > \Sigma$ over all votes
- write/read - overlapping rule:
  $Q_R + Q_W > \Sigma$ over all votes

The write/write overlapping rule ensures that parallel write operations over one copy will be synchronized. If in addition at least Q_W copies will be updated synchronously with the transaction-commit, an update operation will always process on the current object value. That together with the write/read overlapping rule guarantees that there is at least one current copy in every read operation quorum.
To determine the current copy(see 2.1 on page 3) you can use timestamps or version numbers. The use of timestamps brings the advantage that the write/write overlapping rule is no longer needed because the update transaction will be synchronized by using the order of the timestamps.
The amount of necessary votes for a read and write quorum can be chosen as needed. The Majority-Consensus method requires the majority of votes for a read and write quorum ($Q_W = Q_R = N/2+1$)[1]. If you chose a asymmetric

[1] N means the amount of copies/votes and "/" means the whole-numbered division

distribution of the read and write quorums then you can optimize one access operation at the expense of the other.
Using the *weighted-voting method* you can favor one or more copies. If a copy provides a higher degree of availability than all the other copies you could assign 2(or more) votes instead of 1 to it. That improves the possibility to reach a quorum if there is an even number of copies. For instance: If you replicate an object to 4 nodes you need at least 3 votes. If there is a very reliable node you can give 2 votes to it. To reach a quorum a additional vote from another node is now enough.
Voting methods are more expensive compared to absolutistic methods in a error free scenario because the communication work to get quorum is very high. On the other hand voting methods do not rely on a single copy (*Primary Copy* section 2.2.5) anymore as it is for absolutistic methods which is important if errors occur. The vote of a lost copy can be replaced by a vote of another copy without applying any replacing algorithms.

Quorums can be structured and unstructured. For unstructured quorums there can be any copy used for a quorum in contrast to the structured case of a quorum. In the case of a structured quorum, copies are ordered in a logical tree or a grid structure. To get a quorum s votes must be gathered in each of l level along this structure. The advantage of these methods is that less copies must be accessed compared to the unstructured case. This can be very important if the number of copies is high.
Note that quorums can be dynamic or static. This becomes an issue if a database gets partitionized and there are not enough copies to reach a quorum. To handle this problem the size of a quorum can be dynamic. In [1], the following dynamic methods are addressed, for example:

- Dynamic-Voting-Method
- the Tree-Quorum-Method with reconfigurable tree structure

2.2.2 Read-One-Copy Based Methods

Read-One-Copy based methods can be seen as a special case of voting. Here, it needs only the vote of one copy for a read operation ($Q_R = 1$). This copy can be the local copy which means a big performance win compared to the other voting methods.

2.2.3 ROWA (Read-One-Write-All)

This is a read-one based method. It is the basis of all Read-One-Copy based methods. Write-All means that all of the copies will be updated synchronously. This will be a problem in the case of network losses or database partitioning and nullifies the advantage of local read operation.
Other approaches try in case of non-availability of any copy to update only the available copies or switch to another replication method. One of these methods is the *Read-One-Write-All-Available-Copies method.*
Read-One-Copy methods are useful if the amount of read transactions is high whereas failures are rare.

2.2.4 ROWA-A(Available)

This is a read-one base method too. It improves the availability for write operations of the ROWA approach in case of network failures. As the name of this method implies, this method will update all copies that are currently available.
This method is only useful if failures are seldom. In case of failures this method will invoke hard management work.

2.2.5 Absolutistic methods

The *Primary Copy* method is a typical representative for these strategies. One determined copy (primary copy) realizes the synchronization of all the other copies. When a transaction wants to update an object, it needs the authorization of the primary copy. The primary copy will also mostly be the only copy that is updated synchronously. So for every update this copy must be accessed, which means that every update is based on the current object value.
Reading operations mostly don't need to access the primary copy because they don't update a object value. Then the best way to read is using the local copy, that should give the best performance . But this could mean that the reading transaction does not read the most current object value because of asynchronous updates. If read transactions access several different objects it could discover data inconsistencies. This can happen if the update of a write transaction was applied to only parts of the read objects.
Absolutistic methods in that case does not guarantee one-copy-serializability for read transactions.
If you need one-copy-serializability for read transactions too then read transactions can be treated as pseudo-update transactions. But these will

eliminate the advantage of local reading.

The availability of an logical object depends strongly on the availability of the primary copy. If the primary copy in not available another primary copy can be determined. To avoid data inconsistencies it must be ensured that there is only one primary copy for each object at any time. Another problem of this approach is that every update transaction needs to access the primary copy. If there are many update transactions this copy could become the bottle neck of the network. The attractiveness of this method is that it is quite simple to realize.

2.2.6 Summary

When the question comes up which method to use in order to synchronize the copies then you should consider the availability and performance of the systems components. For instance, it is better to use a absolutistic method if you have a powerful mainframe with 100% availability, but a voting approach may be better for smaller, less powerful desktop computers with a higher possibility of failures. In such an environment the possibility to lose the primary copy is much higher than in the mainframe environment. And it is important to distribute the replication work to several different computers in the desktop environment in order to avoid bottle necks. That's why a voting approach is more suitable in those configurations.

2.3 Net Partitioning

The availability of data is an important reason for replication. That's why it should be possible to access the data even in the case of net partitionings or node losses. The simple case is if a node is not reachable, then the update will be processed over it later on. Partitioning of the network is more difficult. Most of the replication methods can tolerate this. There are two general assumptions.

2.3.1 Optimistic Methods

These methods assume that inconsistencies are seldom. If there are any, optimistic methods assume that these inconsistencies can be resolved. That's why it is not necessary to deny access to data in different partitions. Optimistic methods differ in how to resolve the current inconsistencies. One representative is a semantic method, the *Data-Patch-Method*. For that you

	Copy Update Methods		
	Absolutistic	*Read One based*	*Voting*
Representatives	Primary Copy	ROWA	Majority Consensus
Synchronous updates	1	all (if possible)	write-quorum
Availability	depends on availability of the Primary Copy	high for read-transactions/ low for write-transactions	high (because does not depend on one special copy)
Administration work	low/high(if errors occur)	low	high

Table 1: Syntactic Copy-Update-Methods

have to define rules for conflict resolution on designing the database. Optimistic methods are asynchronous replication methods.

2.3.2 Pessimistic Methods

These methods assume the worst case. If inconsistencies in net partitions can occur than they will occur. That's why it is better to reduce access to the data in order to ensure data consistency. Updates over an object is only allowed in one partition. Read access to other partitions is allowed. To join the partitions afterwards will be no problem because updated objects reside in one partition only. Those can then be synchronized to other partitions. All the copy-update methods discussed in this chapter (section 2.2) are pessimistic methods.

3 IBM's Data Propagator Relational

The need to provide easy and rapid access to vital data is a major issue with most companies today. In a typical situation, data is kept in a single location, and users access the data either remotely or locally. This requires that they have access to the database server where the data is located when they access the data. This database access is not always possible. As an example, take the growing base of mobile users with laptop computers who require access to company data on the road. They do not have the ability to connect to the corporate database at all times.
In some situations you may choose to have your own local copy of the data that you need. This is sufficient as long as the original data do not change. If the original data is not static, then you have to worry about keeping your own local copy up-to-date with the master copy of the data. This can be a tedious and time consuming exercise if you have to make a new copy of the required data on a regular basis.

You may find that you need to have a history of the changes that have been made to your data over time. In this situation, a point-in-time picture of the data will not meet your needs.

With IBM's Data Propagator Relational (DPropR) you can handle multiple copies of data. Copies can be generated once or may be resynchronized on an on-going basis.

The contents of this chapter are based to a large extent on [6].

3.1 Classifying DpropR

DPropR is an log-based change-data replication product that updates the replicas asynchronously. It features update-anywhere replication and comes with collision detection and transaction compensation features.
This section tries to classify the mechanisms DB2 DpropR is based on. It can not be classified in any of the replication methods discussed in chapter 2.2 on page 5 because DpropR is an asychonous replication tool which does not meet the requirement that at least one copy must be update synchronously to ensure data consistency (section 2.1 on page 3).

3.1.1 Synchronous–Asynchronous replication

IBM's Data Propagator Relational is an asynchronous replication tool. With an asynchronous replicator, the source update is independent of the replication process while synchronous replication updates at least one replica synchronously. What methods for synchronous replication are available to update all copies and to ensure that all replicas are consistent is discussed in chapter 2.2 on page 5.
With asynchronous replication the users transaction completes when the local update is completed. The replicator updates replicas only after the user transaction commits the changes to the local database. The replication of the updates made by the user may occur moments after the source transaction, in near real-time, or it may be scheduled for later execution. In case of DPropR it can be continously, a minute, an hour, a day, a week or even longer.
With synchronous replication a transaction only commits if the copy or several copies are updated synchronously. It will not commit if a single commit of any copy is missing.
The benefit of asynchronous replication is that it minimizes the impact on the user transaction and increases the robustness of the replication process. Users can continue with other work as soon as the local database applies the update, they do not have to wait for updates to be applied to the remote replicas.
As you can see, these are very practical reasons to use asynchronous replication. But unlike synchronous data replication asynchronous data replication does not retain the four properties – *atomicity, consistency, isolation and durability* – of the principles of ACID [4] and therefore it does not ensure data consistency. For instance, there is no isolation of transactions with asynchronous replication. Transactions could run in parallel and update an object without any guarantee that it updates the most current object value. This problem known as *lost update anomaly*[4].

Benefits of Asynchronous Replication:

- With less processing attached to the user transaction, the user regains control of the system sooner.
- Users are not effected by network delays or slow remote processors.
- User can continue work even in the event of network or remote database outages.

Benefits of Synchronous Replication

- All replicas remain synchronized and that means consistent. Synchronous replication retains all four properties of ACID.
- The use of two-phase commit eliminates the possibility of data collisions.

Latency
Asynchronous replication introduces a period of latency. This is the time after a user applies an update to an object locally and before the replicator applies it to all the replicas. The duration of this time can be defined for the *Apply* process for IBM's DPropR.

Conclusion
There are a lot of reasons in the real world for using asynchronous replication, especially if you think of scenarios with occasional connected mobile replicas which only allows asynchronous replication.
And than there is the performance point of view which is an crucial aspect for the customers. In [3] Robert Goldring said:

> " We must go back to the basics and understand that concurrency control mechanisms have a role in transaction processing. A centralized DBMS will run faster if locking is – disabled, but the results are not – desirable. This, we understand. An asynchronous update replication system is a lot like DBMS that is run without locks – it runs more freely than a system using two-phase-commit and multi-side update, but there are problems."

3.1.2 Mechanism (Log-File – Trigger)

Asynchronous replicators can be either log-based or trigger-based. DB2 Universal Database's data replication function features log-based change-capture technology.
A log-based replication mechanism scans the log record maintained by the DBMS to find changes of data in registered replication tables. The changes get then captured and replicated to the target tables.

A trigger-based replication mechanism instead embeds code in database triggers. The database executes these triggers when a user changes data registered for replication. In theory, a trigger-based replicator could be synchronous, with the trigger code assuming responsibility for updating the replica.

Since log-based replication is fully asynchronous with users transaction, there is a period of time after a users update before the replicator becomes aware of the update. This increases the complexity of collision detection for symmetric replication (source and destination tables are updateable) .

Benefits of log-based replication

- All "industrial strength" DBMS's maintain logs, but not all DBMSs provide triggers.
- None of the replication process is attached to the user transaction, hence the user regains control of the system sooner.

Benefits of trigger-based replication

- Vendors usually guarantee upward compatibility of triggers as they introduce new versions, whereas most vendors do not promise to maintain the structure of their log record across different versions. This is because the log record is considered to be an internal component.
- There is usually small delay between completion of the source transaction and updating of replicas.
- The collision detection process is normally simpler.

3.1.3 Copy-Update method

As mentioned above, IBM's replication product DPropR works asynchronously. Thats why it's not possible to compare it with the copy-update strategies described in *chapter 2.2*. In *chapter 2.2* we discussed different methods and problems of synchronous updating of replicas. Synchronous replication ensures data consistency at any time. This is the main argument for using synchronous replication but it is often not realizable in the real world (see section 3.1.1).
With DPropR all the replicas you have defined as replication targets will be updated later on. Data consistency cannot be assured in update-anywhere replication scenarios. User could update the same object at the same time or not even using the current objects. Instead you need to detect the inconsistencies and compensate them. This is a feature what DPropR offers.

3.1.4 Collision detection

It is not possible to cover all problem scenarios with conflict resolution routines. Problems with asynchronous replication are fundamental in it's nature.

"Researchers recognize the futility of using conflict resolution routines to enforce database correctness:

> "With each database there is – associated a set of integrity – constraints. A database state is consistent if it satisfies the integrity constraints of the system. Thus, if all the integrity constraints of the system could be explicitly specified, then correctness could be – ensured, without resorting to – serializability, by the continuous monitoring of each transaction to determine that it sees only consistent database states and that its execution does not result in a violation of database consistency. There are, however, numerous problems with this approach. A system that continually checks if database consistency is preserved will have poor performance. Depending on the integrity constraints the above task may not even be computable. The – ultimate argument against this approach is that, in practice, not all the integrity constraints of the – database can be explicitly – specified".[2]

The bottom line is: If you rely on automatic conflict resolution routines, you will not be able to successfully audit your transaction executions. If you do not compensate incompatible transactions, and their descendants, your replicas will contain persistent – inconsistent."[3]

DPropR comes with the features of automatic conflict detection, automatic transaction compensation and automatic compensation of dependent transactions for an update-anywhere scenario. DB2 DataPropagator provides three levels of conflict detection: no detection, standard detection, and enhanced detection. Each level has a numerical value which is stored in the CONFLICT_LEVEL column of the register control table. You must decide, based on your tolerance for lost or rejected transactions and performance requirements, which type of detection to use.
If you want to handle those conflicts you will need a global repair strategy. This task is left to the application developer.

3.2 Overall architecture

To setup a replication environment you need a data replication source and a data replication target. You need to determine the source and target tables.

One source table can be used as a replication source for several target tables which can be tables in different databases on different servers. Target tables must not have a 1:1 relationship to the source tables. They may be enhanced as compared to the format of the associated source tables. Replication can "transform" data from the source table to target table in several ways:

- Filter out or subset rows, from the source data to a more meaningful amount
- Filter out columns of sensitive data which are not appropriate for this given user
- Columns in the target table may be derived from, or calculated on, columns in the source table by using SQL aggregation functions. This is applicable for *base aggregate* tables (see section 3.3 on page 16) only.

Once the source and target tables are specified you can start the replication process. This includes two processes. It is **Capture** and **Apply** which perform the replication of the data from the source to the target side(see Figure 1 on page 16). The *Capture* process is running on the source server and determines which changes(updates) were made against the replication source and writes these rows to special tables, the control tables, which contains all the changes of the source table. After *Capture* has done it's job the data are available to be replicated to the target. When Apply gets invoked it will read these changed data from the control tables and copies it to the target tables.

3.3 Replicated Data

Data can be copied from the replication source tables to the replication target tables differently. That depends on the type of target table you want. Target tables can be read only or updateable. There are several types of target in DB2's DPropR tables:

- *User copy (read only)*
  This is a complete, consistent copy of the replication source table. It must have a primary key. It can be a subset by row(*selection*) or[2] by column(*projection*).
- *Point-in-time (read only)*
  This is a complete, condensed copy of the replication source table at

[2]"or" means logical an or not(!) an exclusive or

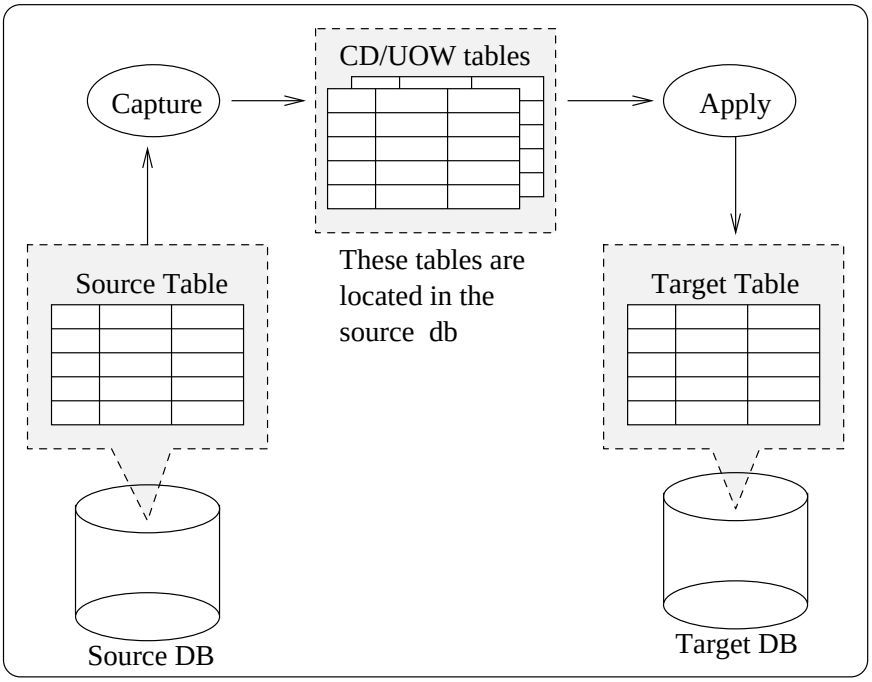

Figure 1: Overview of a replication process

a certain point in time. It must have a primary key. This table is enriched with a timestamp column to indicate when a transaction on the source table occurred. It can be a subset by row or by column.

- *Base aggregate (read only)*
  This is a history table in which new rows are appended for each subscription cycle using the result of an SQL column function calculation against the replication source table data. This table summarizes the contents of the source table itself.

- *Change aggregate (read only)*
  This is a history table in which a new row is appended for each changed row in the replication source table using the result of an SQL column function calculation against only recently changed data. Basically, you'll find all the changes aggregated in that table.

- *Consistent change data (CCD) tables (read only)*
  These tables are used as staging tables. Changed data is copied from the replication source to the target tables. CCD tables only consist of committed data. Once data was copied to the CCD tables it can be

used as a source in a replication scenario with other target tables. This concept makes an environment more flexible and efficient.

  - *Condensed, non-complete*
    This staging table contains the most current values of updated rows. Only rows which were changed can be found it that table.
  - *Condensed, complete*
    This staging table contains all the rows from the replication source and the most current values from each row.
  - *Non-condensed, non-complete*
    This table is intially empty and will be appended by each insert, update and delete action on the source table.
  - *Non-condensed, complete*
    This table is initially a complete copy of the source table and will be appended by each insert, update and delete action. It contains the whole history of the source table.

- *Replica (updateable)*
  This table can be updated and is used for update-anywhere scenarios. Changes on the target table got replicated to the source table.

3.4 Capture

Captures task is to determine all the changes that were made against the replication source and makes those available for replication. There is only one single *Capture* process for one database and it will act on all registered source tables in this database. Therefore it runs on the source server. Typically *Capture* runs continuously, but you can stop it while running utilities or modifying replication sources.

When capture gets started it reads the register table to determine the source tables for which it needs to start capturing changes. DB2 records all transactions in log file for recovery and diagnostic. *Capture* reads this log file to detect changed records from source tables that are defined as replication sources. It does not need to access the source tables for capturing changes. It will place changes to the Changed Data (CD) tables. There is one CD table for each registered source table. *Capture* also retrieves information about committed changes and stores it in the Unit-Of-Work (UOW) table. The rows in that table identify those transactions that have been committed. This happens in the same order as it can be found in log record, it preserves

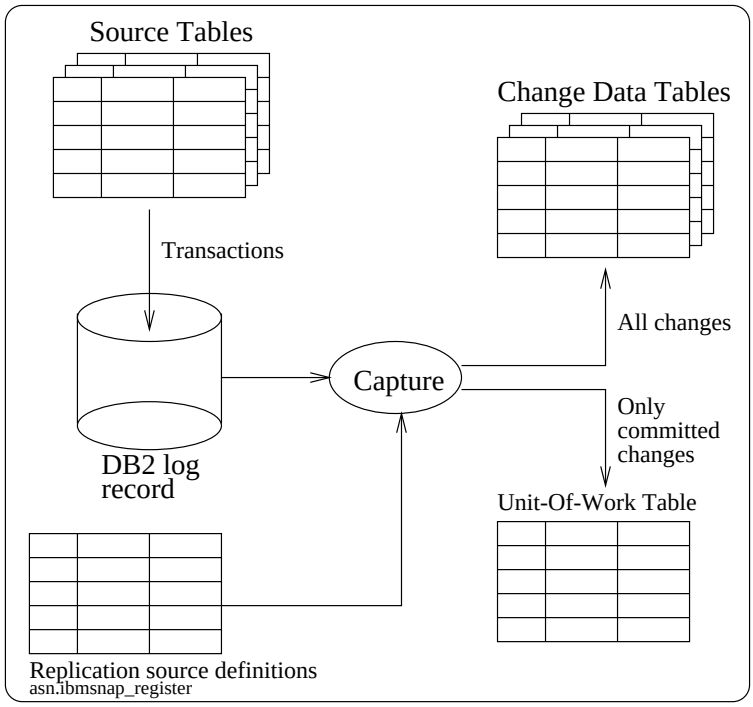

Figure 2: Architecture of *Capture*

the order of transactions. That ensures transaction consistency for the replication process. Then it updates the register table to record the amount of committed data that was captured for every replication source.

3.5 Apply

Apply is the program that copies the changed data from the replication source to the replication target. It does that by reading the changes stored in the CD table and applies it to the target tables at local or remote servers. *Apply* can also perform column functions on the data from the source or CD tables and appends these results to the target tables. It copies data either by full refresh or by differential refresh.

For a full refresh *Apply* does not access any CD or CCD table. Instead it will directly access the source table and copy it entirely. The *Capture* program is not involved at all, it does not capture any changes and write to none of the control tables. Full refresh is used for the initial load of the target table. You can specify full refresh when you define the replication

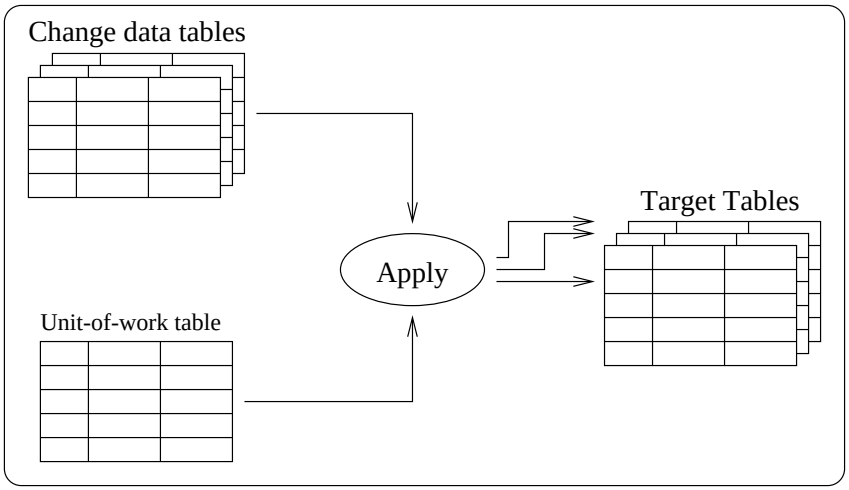

Figure 3: Architecture of *Apply*

source.

For differential refresh, *Apply* copies only changed data from the CD (in conjunction with the data in the UOW table to replicate only committed changes) tables to the target tables. When *Apply* gets invoked the very first time or after a cold start of the *Capture* program it does a full refresh to populate the target database. From then on *Apply* uses differential refresh.

3.6 Database Protocol Based Connection

Distributed Relational Database Architecture (DRDA) is a standardized database protocol that provides reliable, fast, and secure access to data within heterogeneous client-server environments. DRDA defines the commands, data formats, and rules that allow any DRDA client to inter-operate with any DRDA server. It allows application developers to use SQL statements in their code.

DRDA provides a robust connection between DB2 clients and servers. The DB2 data replication product DPropR is as the entire DB2 family DRDA enabled. Database applications or tools that are coded using standard SQL APIs as embedded SQL can automatically access remote databases because the underlying database client code in DRDA-enabled. So the application does not need to know about the location of the database in the network and is free from communications code needed to access the database, which

is know as *Distribution Transparency*.

4 Overview of Data-Links

Data-Links is an innovative software that has broad appeal across a variety of application segments. Any application with significant content-management requirements can benefit from Data-Links, from Web-based electronic-commerce and intra-net applications to more traditional computer-aided design and manufacturing (CAD/CAM) applications which created the initial customer push for Data-Links.
Data-Links meets a very challenging application requirement that has existed for many years. It enables organizations to continue storing data (particularly large files of unstructured or semi-structured data such as documents, images, video clips, and engineering drawings) in the file system to take advantage of file-system capabilities, while at the same time coordinating the management of these files and their contents with associated data stored in an RDBMS.

The contents of this chapter are based to a large extent on [5].

Data-Links gives customers comprehensive control over external data in the following areas:

- *Referential integrity*
  Data-Links ensures that users cannot delete or rename any external file as long as it is referenced in the database.

- *Access control*
  DB2's permissions can also be used to grant the ability to read a referenced external file. Read access control is optional it can be left to the file system or given to the DBMS.
  Write permissions can be left with the file system or blocked. "Blocked" means that a referenced file cannot be updated in place; a a new version must be created and then the link switched in the database through a SQL "update" statement. If the file system retains write permission, files can be updated in place. In this case, there is no "update" statement issued against the DB2 table. Therefore, DB2 cannot support coordinated backup and recovery because it is not informed about updated files.

- *Coordinated backup and recovery*
  The DBMS is responsible for backup and recovery of external data in synchronization with the associated database. This type of control over external data is also optional.

- *Transaction consistency*
  Changes that affect both the database and external files are executed within a transactional context to preserve the logical integrity and consistency of the data.

In addition to the advantages inherent in RDBMS management of external data, Data-Links have some significant benefits. It,

- Allows external files to be stored in close proximity to the application to reduce network traffic and maximize application performance.
- Maintains speed of file access by continuing to use the file system for this, not the DBMS. This is also critical to application performance.
- Requires minimal or no changes to existing applications.
- Works with any file system on Unix or Windows NT, and takes advantage of support for hierarchical storage managers within the file system.

4.1 Overall architecture

Data-Links consists of several components:

- The Data-Link datatype – This is a new build-in base type of DB2 UDB
- The DB2 Data-Link Manager software on the file server – The Data-Link Manger software consists of two components: the Data-Links File Manager (DLFM) and the Data-Links File system Filter (DLFF).
- DBMS/DLFM APIs which is used to communicate with the Data-Links Manager on the file server.

The DB2 UDB and the Data Links Manager can run on different platforms. The Data-Links File Manager itself can run on different multiple, heterogeneous file systems. So the database can contain references to files stored in multiple, distributed file systems. That allows the files to be close to their applications, reduces network traffic and improves application performance.

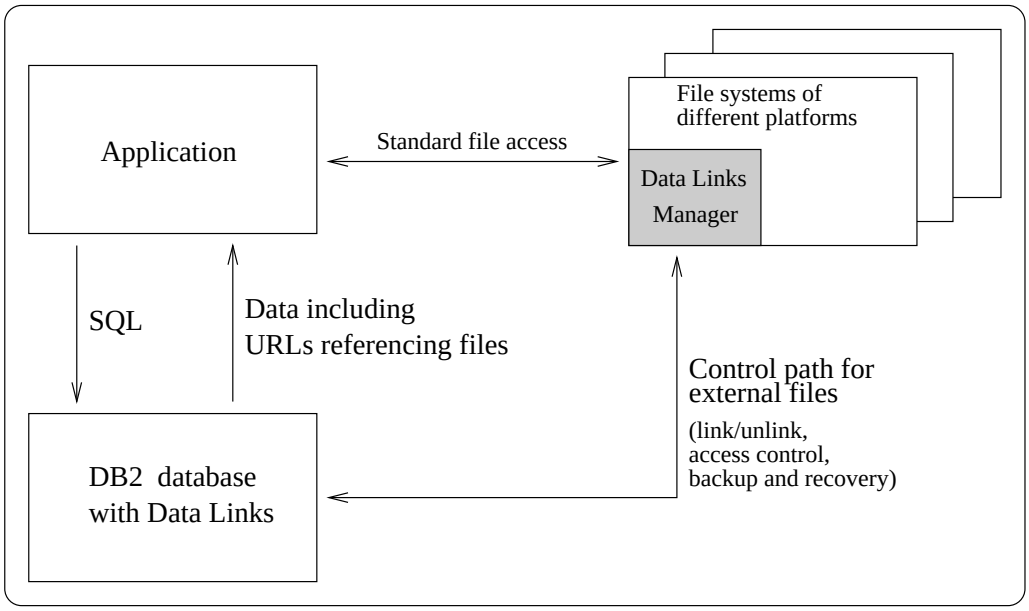

Figure 4: Overview of Data Links Manager

4.2 Functionality

The Data Link File Manager(DLFM) plays the key role in managing external files. It is responsible for the link and unlink operations with transactional semantics within the file system. The DLFM maintains its own repository for the linked files. When a file is initially linked to the database, the DLFM applies the constraints for referential integrity, access control and backup and recovery as specified in the DATA LINK column definition.
Before an external file can be linked to a database, it must be created by an application. Afterwards a reference of this file which is an URL, for instance *http://sample-svr.com/sample-dir/sample-file.txt*, can be inserted in a DATA-LINK column of a table. Applying SQL statements for modifying DATA-LINK data will provoke DB2 to do the following:

- INSERT statement:
  The insert statement prompts DB2 to run a Link-File operation. This Link-File operation will be done by the Data Link Manager. From then on the DBMS got control over the file.

- DELETE statement:
  A delete statement will prompt DB2 to unlink the file and delete the row. The file can now be deleted or returned to control of the file

system.

- UPDATE statement:
  An SQL update statement is the result of an Unlink-File operation for the old URL and a Link-File operation for the new URL.

The DLFM is also responsible for "garbage collection" of backup copies of unlinked files that are no longer required by the DBMS.

The DLFF is a thin, database-control layer on the file system that intercepts certain file-system calls, e.g. file-open, file-rename or file-delete issued by the application. It ensures that any access request meets DBMS security and integrity requirements. The DLFF does also validating any authorization token embedded in the pathname for a file-open operation.

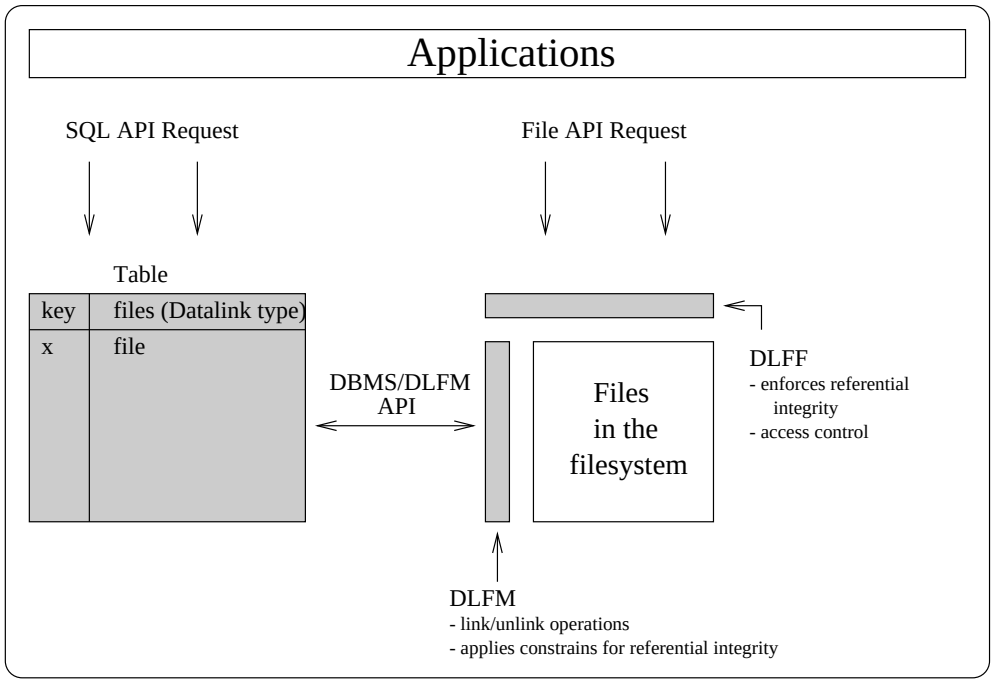

Figure 5: Architecture of Data Links Manager

5 Description of The Problem

Data-Link data consists of two pieces of information:

- the logical reference to an external file (stored in the database)
- and the physical copy of the data file (stored at the file server).

Therefore, when replicating a Data-Link value, we need to replicate the information mentioned above. The replication order should be:

- we need to copy the data file first to the target site
- and then the Data-Link value, so that the database can link (reference) the file. Linking a file requires the existence of the file.

For doing that, we have two major technical issues to be addressed:

1. how to transfer the data file from the source file server to the target file server.
2. how to know where (at which file server and in which directory) the data file will be stored at the target (i.e. mapping between the source data file to target data file, discussed in section 6.3.5 on page 31)

Given the existing architecture of Capture and Apply, this is not an easy task.

5.1 Data Links And DRDA

The DB2 replication tool DPropR propagates only data between different databases using the DRDA protocol.
All of the data to propagate are existing in some databases within a distributed database environment. Therefore, the replication tool uses the DRDA protocol to copy the appropriate data between different databases because those databases are able to communicate with each other using the DRDA protocol.
For Data Links the case is different. There are data inside the database and outside the database as well. The data outside of the database are stored as files in the file system and not in the database itself. Only logical links to those files are stored in tables inside of the database and are managed by the DBMS. That means that we can use DRDA to replicate the file references inside the database as usual as any other datatype(VARCHAR), but there exists no solution which gives us the opportunity to use DRDA to replicate files existing in the file system to different servers.

5.2 Data Links And The Log Record

DProprR is a log-based replication technique. It looks in the log record for changes made to the registered source tables. But with Data-Links only the references to the files in the file system are stored and managed in the database. Therefore, there will only be the reference in the log record but not the file itself. All what Capture can do is to capture the Data-Link(reference to a file) found in the log record and copy it to the changed data table.

5.3 Data Links And Apply

The same problem comes up with the Apply program. Apply does not know how to copy a Data-Link-ed file. Since the DRDA protocol does not know the datatype file and how to access the file system in order to read and to write a file there will be no way for Apply to replicate file data between distributed database systems.
The only way for accessing a file referenced in a database would be accessing it directly using standard I/O of ftp. This is what we discuss later on in chapter 6 on page 27 for the prototype design.

6 The Prototype

6.1 Idea

The requirement for this prototype is not to change the existing code of *Capture* and *Apply*. The idea is to use the existing ability to replicate the BLOB datatype of DPropR. The file data should be converted to a BLOB and inserted into the database. Afterwards it can be replicated using a DRDA connection. On the target side the BLOB must then be materialized back into a file again.
When inserting the Data-Link value into the target table the file will already exist at the target file server and the linking can be done.

6.2 Design architecture

This prototype addresses the file transfer issue without modifying any existing component. We will also address the second issue, the file mapping, here and provide a simple solution for it.

There are several pieces added on the top of the existing components:

- Split the base table into two. One with no Data-Link columns and the other with a foreign key column and all the Data-Links columns. The table with no Data-Link columns will be defined as replication source for the *Capture* program.

- A join view to put both tables together. Instead of defining the view on the Data-Link columns, we define the columns as a UDF of a Data-Link which returns a BLOB containing the corresponding file and the reference string in front of it. This view will be used by *Apply* as a source table. When *Apply* is going to propagate the data it will copy this BLOB instead of just the reference to a file.

- A join view to add a "char(1)" column for each Data-Link column of the base tables to the CD table. This view is defined as CD table for the *Apply* program.

- For each target table, two extra columns are defined. One is for storing the BLOB value. The other is for the actual Data-Link column.

- Two file mapping tables will be defined within the target database to describe where to store the file.

- An after-SQL stored procedure is defined at the target server to first identify the file location and then convert the BLOB value to a file. Finally, it updates the Data-Link column into the target table. The stored procedure will be called by *Apply* and must be defined in the subscription.

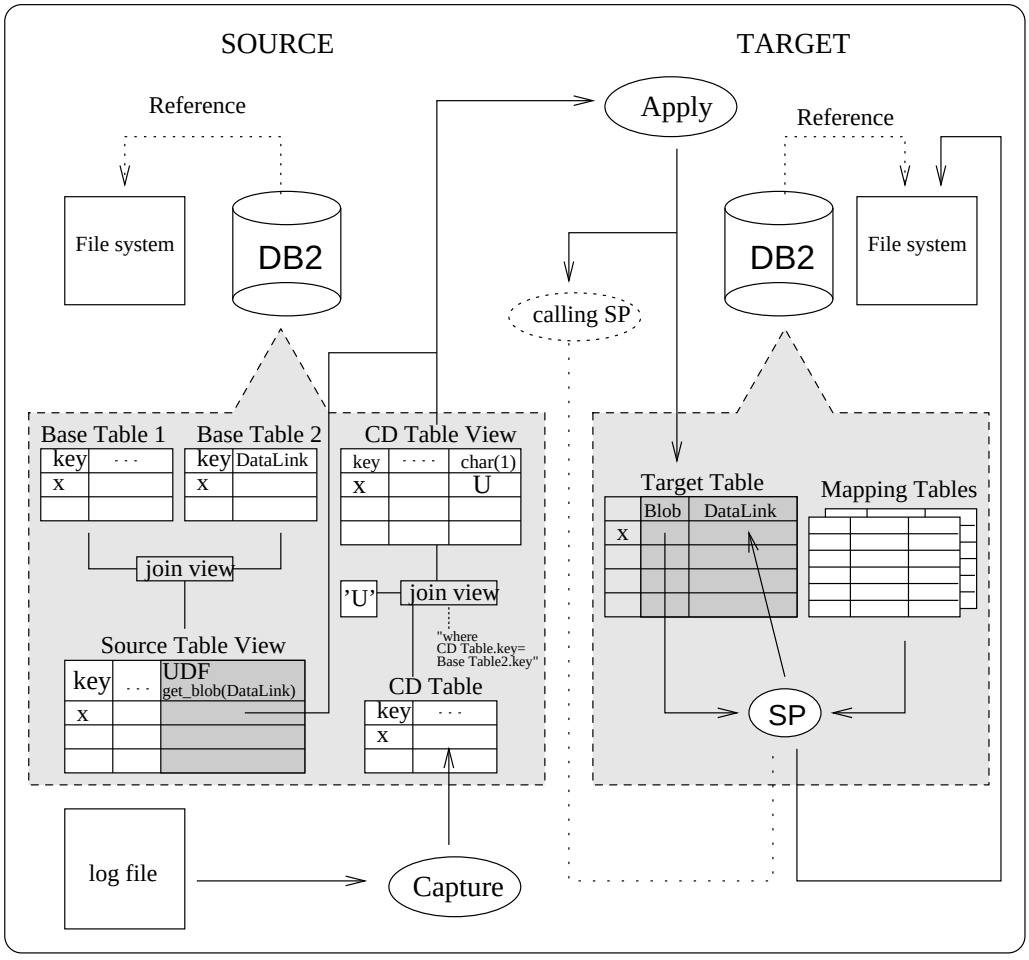

Figure 6: Architecture for the Data-Link replication prototype

The control flow:

1. *Capture* reads the log records for updates in the replication source ("Base Table 1").

2. For each log record, *Capture* inserts a row into the CD table.

3. When it is time for *Apply* to replicate the data, it reads the changes from the CD table defined in the subscription. This is the "CD Table View"-table.

4. *Apply* will read, beside the data from the other columns, the indicator column in "CD Table View"-table and finds an 'U'. The 'U' means 'Update of a LOB datatype'. This tells *Apply* to access the source table "Source Table View" directly to retrieve the data for propagation.
   It gets the data from a column which is defined as UDF of a Data-Link. This UDF returns a BLOB in which it wrote the Data-Link itself and the content of the file.

5. *Apply* will then replicate the BLOB value as well as the other changed columns to the target table.

6. After *Apply* has replicated all changed data it calls the stored procedure.

7. The stored procedure first converts the Data-Link value from the source server to the target server depending on the contents of the file mapping tables.(for mapping see section 6.3.5 on page 31).

8. Then it copies the BLOB value into a file to its new location at the target side.

9. Finally, it updates the Data-Link column in the target table which establishes a link to the file at the file server.

6.3 Internal Design

This section is going to describe each individual add-on component in more detail.

6.3.1 Using UDF for The File-to-BLOB conversion UDF

A UDF function will be defined at the source server to perform the following tasks:

- retrieve the content of the file specified by a URL value
- then convert them into a BLOB format and return it to the caller

It takes VARCHAR as an input parameter and BLOB as the return parameter.

6.3.2 Retrieve the file content

There are several ways to retrieve a file:

- If the file is located at the same machine (either on a local or network file system), using the standard I/O file function will be sufficient if DB2 has the permission to access the file.
- For a remote file, we could use the HTTP protocol but with a requirement that there is a HTTP daemon running on the remote side and the daemon has the permission to touch the file.
- We can also use the FTP protocol. It requires a user and a password to be set up and known by the UDF to be able to retrieve the file. The user/password information can be specified for each registered file server in the Data-Link configuration file. The same FTP protocol would also be used to store the file on the target side.

6.3.3 File-to-BLOB conversion

The conversion will be simple. First, the UDF creates a BLOB object and stores the file name in the beginning followed by a zero bit. Then it opens the file and appends it's content as a buffered stream to the BLOB object. We put the file name at the beginning of the BLOB object so that when we materialize the file at the target side, we will know the original Data-Link value to do the file mapping.

This approach does not handle the case where the file representation differs between the source and target side.
Since the size of Data-Link files could be large, to reduce the amount of data transferred over the network, we should consider to compress the data before returning it as a BLOB. There are many well-accepted compression algorithms that we can use. However, it usually requires the file to be materialized before compression can take place, rather than compress on the fly. Therefore, a "tmp"-directory is required to store a copy of the file. Since compression requires additional disk I/O and CPU time, we should only perform the compression if the file is bigger than certain size.

6.3.4 Data in CD table

The current version of *Capture* does not recognize the Data-Link datatype in the log record and will crash if it finds any. That's the reason why we must not have a Data-Link column in the replication source. We had to

work around that problem and found a solution to make it work.

Because we need to define a Data-Link column we split the source table into two tables. One table contains all the original columns except the Data-Link column. This table I call *"base table 1"* and will be defined as replication source for the *Capture* process, so that capture does not get in touch with any Data-Link column. The second table will contain the primary key columns plus the Data-Link column. That I call *"base table 2"*. Since we defined a view over these 2 base tables we can now create a CD table that has an extra column for an updated LOB value.
To achieve that we define the CD table as a join view over the original CD table and *"base tables 2"* which contains the primary key columns and the Data-Link column. But instead of adding the Data-Link column itself we add an char(1) column and insert an capital u ("U").

The definition of the table looks like that:

```
CREATE VIEW GUIDO.CD-Table2 as select
            a.IBMSNAP_UOWID, a.IBMSNAP_INTENTSEQ,
            a.IBMSNAP_OPERATION, a.deptno, a.deptname,
            a.mgrno, a.admrdept, a.location,
            substr('U',1, 1) as dlblob
FROM GUIDO.CD-Table1 a, GUIDO.deptdltbl b
            WHERE a.deptno = b.deptno;
```

When *Apply* is going to copy the changed rows and finds the column with an U, it will copy the BLOB value by accessing directly the replication source table which is for apply the join-view over base table 1 and the UDF of the Data-Link column from base table2.

6.3.5 How to do the server-to-target file mapping

A Data-Link value composes of three basic elements :

- the server name
- the prefix name
- and the stem-name

The Server name is the host machine which contains the file. The Prefix name is the mount point of the Data-Link file system and is defined in each Data-Link File Manager(DLFM). Finally, stem-name is

the rest of the value. For instance, if we have a Data-Link value, *http://dl.stl.ibm.com/ shared_dlfs/x/pict.gif* , the server name will be "dl.stl.ibm.com", the prefix name will be "/shared_dlfs" and the stem-name will be "x/pict.gif".
Therefore, our first approach is to have two mapping tables in each target database. It will be one table for the server mapping called:

- IBMSNAP_SERVER_MAP
- IBMSNAP_PREFIX_MAP

The IBMSNAP_SERVER_MAP table will be used to map the source server, to a server connected to target database. For example, if the source URL is *http://source_svr.stl. ibm.com/path/file*, the (server-) mapped URL would look like *http://target_svr.stl.ibm.com/path/file* where source_svr.stl.ibm.com is the file server on the source side and is mapped to target_svr.stl.ibm.com.
We also plan to have a default value in the table so that all the servers that are not inserted in IBMSNAP_SERVER_MAP table are automatically mapped to a default target server.
If the table contains neither the mapping for the source server nor a default value, an error will be returned.

A IBMSNAP_SERVER_MAP table could for instance look like the following:

Server Id	**Source Server**	**Target Server**
1	src_test1.alm.ibm.com	trg_test1.stl.ibm.com
2	src_test2.alm.ibm.com	trg_test2.stl.ibm.com
3	src_test3.alm.ibm.com	trg_test1.stl.ibm.com
...	...	...
default	any other server	trg_test1.stl.ibm.com

Table 2: Sample of IBMSNAP_SERVER_MAP table

The IBMSNAP_PREFIX_MAP table will be used to map different prefix names included in the source Data-Link value to different paths on the target server. For example if the source URL is *http://source_svr /a/b/file*, the mapped URL will look like: *http://target_svr/x/file* where the path /a/b is mapped to the path /x. It also supports default directory. If no

mapping can be found for the source URL, the file will be replicated to a default directory. No mapping will be performed on the stem-name.

We also thought about the problem when the file that we want to replicate already exists. We decided for the solution to rename the file and to come out with an appropriate message. At this moment, it is not clear how the naming convention for those files will look like. The simplest way will be appending a number to the URL.

A IBMSNAP_PREFIX_MAP table could look like this:

Server Id	**Source Prefix Name**	**Target Prefix Name**
1	/x	/a
1	/test	/a
2	/test	/b
2	/x/y/z	/b
3	/x	/test
...	...	...
default	*	/a

Table 3: Sample of IBMSNAP_PREFIX_MAP table

The advantage of this approach is that if no mapping is defined for a particular server, we would still perform the replication as long as we define a default entry and it is valid. Then customers will be able to find the files in the default directory *server/a*.

When defining a Data-Link column in the target table, we have to be careful about the link options. For example, when a Data-Link column was changed in the source side and got replicated to the target, if the Data-Link column has "ON UNLINK RESTORE", the file will still be there even after the update and we will have two files (the replicated file will be renamed to something else.)

6.3.6 Stored procedure to materialize the file

After the BLOB got replicated to the target table, *Apply* will invoke the stored procedure in order to perform the followings:

- first it does a query over the Data-Link column

- it will select the BLOBs of all the rows which have an empty("null" value) Data-Link column
- afterwards it extracts the URL which is stored in front of the BLOB
- then it maps the URL to the target side using the mapping tables as described in section 6.3.5 on page 31
- the next step is to read the content of the BLOB and to materialize it into the file
- if the file server is at the same host as the database server, it stores the file directly using standard I/O operations. Otherwise, we will use FTP as described in section 6.3.2 on page 30 to send the file over
- the last step is to insert the URL in the Data-Link column of the target table. If this column was defined with "no link control" the Data-Link Manager will do nothing. Otherwise it will do a link operation to the file.

6.4 Limitation and Potential Performance Issues

Regarding to this design, there are several technical problems and performance issues.

Usability issues:

- The current *Capture* program cannot handle DBMS log record with Data-Link column. It will abort if a replication source table contains any Data-Link column. Therefore, to get around the problem temporarily, we create another table to store the Data-Link columns separately along with the primary key column using join-view to retrieve the Data-Link value. We define only the original table as the replication source, not the second one. Therefore, in order to insert a row, it requires first to insert the row to the original table and then insert just the Data-Link values and key to the second table.
- There is a restriction that no input parameter is allowed for any stored procedure used in a subscription. Therefore, you cannot call the stored procedure with any parameter and the target table name, the BLOB column name, the column name which is defined as primary key and Data-Link column name must be hard coded inside the stored procedure.

- The target Data-Link column must always accept null values because the value will not be filled in until the stored procedure is called.
- We did not solve the problem, what is to do if we replicate a file that already exists at the target side, which is also the case for updates of data linked files. The problem is mention in section 6.3.5 on page 33.

Technical issues:

- The original row in the source table may already be removed while we retrieve the Data-Link value.
- Since we do not use the Recovery_ID, the file that we retrieve may not match the value of other columns in the row.
- Since with the Data-Link datatype problems in the *Capture* program, we do not define the second base table (containing only the primary key columns and the Data-Link column) as replication source and changes on this table will not be captured. Therefore, an update on the Data-Link column will not be recognized by the *Capture* program and the replication process will not work if we change the Data-Link column in base table 2.
- This design does not address on how to assign the owner of the replicated file. Currently, it will be owned by the FTP user. We would consider to add an extra column (like Default Owner) to the IBMSNAP_PREFIX_MAP table to store the default owner id of files created under the corresponding directory. Still, the problem is that files would be on a remote machine and we cannot change the file owner through the FTP channel unless we have a root process running on the remote machine to do the job.
- The file replication does not maintain the file permission. For example, on UNIX, in order for a file to be an executable, it must have an execute permission. However, the permission of the target file will be whatever the default value is and will not be the same as the permission of the source file.
- This prototype has also not addressed the issue if the file format of the target server is different from the source server, such as UNIX vs NT.

Performance issues :

- It requires at least three intermediate steps in order to transfer a file from the source file server to the target file server. From source file server to source db, from source db to target db and materialize the BLOB data on the disk on the server side and then from disk to target file server.
- Each lookup by join view requires access to the source table directly. It would interfere the normal operation in the source server. They both requires extra processing resources.
- There is an extra disk storage requirement on the target side to store the BLOB data.

7 Implementation

There were 2 people working on this prototype. It was Joshua Hui and myself. So we split the work into two pieces. Joshua did the work at the source side and I did the target side.
The implementation of the prototype was done in java because there was already a UDF example written in java which converts a file into a BLOB. So, Joshua modified this UDF and wrote an ftp client class that was used by the UDF and Stored Procedure (which was written by myself) the read and write the file from and to remote servers.

7.1 Description of the implementation

The whole project contains several SQL files which are needed to setup the replication environment. It also contains of four java classes. The class Data Channel is to establish a data channel to the remote side either to send or to receive data. The class FTP is a simple FTP client. It is used by the UDF and Stored Procedure to send and to receive the wanted files and to create the necessary directories. The class StP contains the stored procedure.

- **Class DataCannel**

```
public DataChannel(String remotehost, int remoteport)
       throws IOException;
public int localPort();
public void open(boolean read)
       throws IOException, Exception ;
public InputStream inputstream();
public OutputStream outputstream();
public void close()
       throws IOException, Exception ;
```

- **Class FTP**

```
public FTP (String host, String user, String password)
       throws IOException, Exception;
public int sendCommand(String command)
       throws IOException, Exception;
public int readResponse()
       throws IOException;
public void asciiMode()
       throws IOException, Exception;
public void binaryMode()
```

```
        throws IOException, Exception;
public void cwd(String dir)
        throws IOException, Exception;
public InputStream startRecvFile(String filename)
        throws IOException, Exception;
public void endRecvFile()
        throws IOException, Exception;
public OutputStream startSendFile(String filename)
        throws IOException, Exception;
public void endSendFile()
        throws IOException, Exception;
public void check_dirs (String filename)
        throws IOException, Exception;
void   create_dirs (String ppath)
        throws IOException, Exception;
String extractPortAddress(String s)
        throws Exception;
private static String makePortAddress(InetAddress inad, int port);
private static String toUnsignedDecimal(byte b);
```

- **Class DB2Udf**

```
public void getBlob(String dlnk, Blob result)
        throws Exception;
public void start_time() throws IOException;
public void stop_time() throws IOException;
public void set_starttime(long start);
public void set_stoptime(long stop);
public double get_starttime();
public double get_stoptime();
```

- **Class Stp**

```
public void PutFile () throws Exception;
public void map_dlnk (Connection con, URL dlink);
public void insert_dlnk (Connection con,String key)
        throws Exception;
public void  read_dlnk(InputStream in)
        throws IOException;
public void write_file (InputStream in)
        throws IOException;
public URL  get_dlnk ();
public void set_dlnk (String new_dlnk);
```

```
public void start_time()throws IOException;
public void stop_time ()throws IOException;
public void set_starttime(long start)
public void set_stoptime(long stop)
public double get_starttime();
public double get_stoptime();
```

7.2 Performance measurements

Performance is the big issue for this prototype as mentioned in section 6.4 on page 35. It is not acceptable to copy a blob three times is in the performance point of view.
To illustrate the severity of the problem we measured the performance of the prototype. I did three different measurements.

1. I replicated Data-Links referencing to files with a size of 2 MB residing on a remote server to another remote server.

2. Afterwards it I replicated 1MB files residing at the same source server to a remote server.

3. The last test I did was replicating Data-Links referencing 500kB files.

7.2.1 Replication scenario

An Intel PC was used to run the replication test. DB2 UDB 5.2 was installed on that machine. The test used two databases (sample and copydb) which were create on that PC.
The aim was to replicate one table (view) from the database "sample" to the database "copydb". There were 10 rows inserted into the source table. One of the columns was a Data-Link column (with no link control). The Data-Links pointed to a remote AIX server. The task was to establish an ftp connection in order to receive the appropriate file and to copy it into the database as a BLOB. When the blob gets replicated the Stored Procedure will search for rows with an empty Data-Link column. If it finds any it will read the reference from the blob. The reference is stored in front of the file data within the BLOB blob. Then is needs to map the reference to the new location. As described in chapter 6.3.5 on page 31 it is necessary to have two mapping tables with the appropriate entries. Table 4 and 5 show how the mapping tables looks like.

Server Id	Source Server	Target Server
1	dlsmp.almaden.ibm.com	shiloh.stl.ibm.com
2	breeze.alm.ibm.com	shiloh.stl.ibm.com
3	default	shiloh.stl.ibm.com

Table 4: Rows of the IBMSNAP_SERVER_MAP table

Server Id	Source Prefix Name	Target Prefix Name
1	/localfs/jhui/srcdir	/home/guido/Test/targetdir
1	/u/hui	/home/guido/Test/targetdir
1	default	/home/guido/Test/targetdir
2	/localfs/jhui/srcdir	/home/guido/Test/targetdir
2	default	/home/guido/Test/targetdir
3	default	/home/guido/Test/targetdir

Table 5: Entries of IBMSNAP_PREFIX_MAP table

7.2.2 Results of the test

The performance of replicating Data-Links is influenced by many factors. For instance if we receive and write the files via ftp the performance strongly depends on how fast the ftp connection itself is. A slow ftp connection could slow down the entire replication process. That's why I will replicate 10 Data-Link values which all point to files with the same size. The UDF establishes every time a new ftp connection and we'll get more realistic results of the prototype and are able to make a proposition.

To measure the performance of the replication process I will use the *Performance Trace*(trcperf) flag of the Apply program. The time I will get from that trace includes also the time of 10 UDF processing (T_{UDF}), how long it needs to receive the files via ftp. That's why I wrote a routine within the UDF which prints the time for each UDF in a different file.
The times from those files will then be added to $T_{UDF} = \sum_{n=1}^{10} UDF_n$ and then substracted from the apply-process-time T_{Apply} I got from the Apply *Performance Trace*. The result $T_{BLOB} = T_{Apply} - T_{UDF}$ will give me the time the database needed for replicating the blob.
The time for the overall replication process is the Time for Apply T_{Apply} plus the time for the stored procedure T_{StP} to materialize the BLOBs into files ($T_{Total} = T_{Apply} + T_{StP}$).

Table 6 shows the time of the UDF for 10 files in three different sizes.

Measures	Time in seconds		
	500kB	1MB	2MB
1.	4.317	10.134	31.375
2.	3.425	7.621	26.158
3.	3.365	7.922	26.498
4.	3.054	7.241	26.639
5.	3.325	7.26	26.138
6.	3.355	7.311	26.147
7.	3.224	7.601	25.597
8.	3.325	7.331	26.117
9.	3.185	7.47	26.087
10.	3.304	7.361	30.574
Average:	3.388	7.725	27.13
Total:	33.879	77.252	271.33

Table 6: Time for UDF

Table 7 will show you the results from the Apply perftrc printouts. These results include the processing time of the UDF and the replication of the BLOB.

10 files with size of	Time in seconds	
	Apply + UDF()	Stored Procedure
2MB	313.41	44.033
1MB	95.918	26.718
500kB	44.452	21.772

Table 7: Performance of replicating Data-Links

The time for replicating the BLOB is
($T_{BLOB} = T_{Apply} - T_{UDF}$):
2MB : $313.41s - 271.33s = 42.08s$
1MB : $95.918s - 77.252s = 18.666s$
0.5MB : $44.452s - 33.879s = 10.573s$

Total time for replicating the file of is
($T_{Total} = T_{Apply} + T_{StP}$):

$2\text{MB} : 313.41s + 44.033s = 357.443s$
$1\text{MB} : 95.918s + 26.718s = 122.636s$
$0.5\text{MB} : 44.452s + 21.772s = 66.224s$

This time just give you an idea about what you can expect from this prototype.
Unfortunately, it is very difficult to compare these times to any other times of any other replication process. Since we are using nothing really new and were just working around the given code of Capture and Apply. So what it means is, that we are only using all the given features of DB2 and DPropR. But if we look at the measured times for replicating the files which look very high, we see that the problems about performance I mentioned in chapter 6.4 on page 35 became true. But those numbers are not unexpected since we knew that we copy the files three times within the whole replication process. To give an impression about how long the replication process last we can just compare it with an usual ftp access to get the files. In table 6 we have already 10 ftp accesses through the UDF T_{UDF} and the average time to get the file $T_{UDF/10}$. I also calculated just above how much time the entire replication process T_{Total} takes. And if we just divide the whole replication process by ten we then have the average time for replicating just one file $T_{Total/10}$.

size	$T_{UDF/10}$ in sec	$T_{Total/10}$ in sec	Difference
2MB	27.13	35.74	**8.61**
1MB	7.72	12.26	**4.54**
500kB	3.39	6.62	**3.23**

Table 8: Comparing ftp access only and prototype

Of course DPropR does a lot more than just copying the file, but most of the time difference comes from copying the Blob. This is the factor what makes that prototype that slow.

8 Conclusion and recommendations

This prototype was the first attempt to replicate a Data-Link datatype within IBM's DPropR data replication product. The goal was to find a way to replicate data linked files without any changes of the existing code.
We realized that goal and produced a prototype which replicated files referenced in a database's table from one server to another. But the prototype has several limitations, for example we had to hard code a lot of important information the prototype needed such as table and column names which made it not very usable (see section 6.4 on page 34).
One of the biggest problems was to setup all the definitions to register and setup the environment for *Capture* and *Apply*. Since the current *Capture* program did not recognize the Data-Link datatype and could not even ignore it without crashing we had to avoid *Capture* to get in touch with any Data-Link datatype. So, we had to be very tricky by defining the replication sources, CD tables and the description.
We already predicted in the prototype design that it would probably perform too slow in order to use it as a customer product. And this prediction was confirmed as described in chapter 6.4 at page 35 when we did the first performance measurements.
You can split the problems of the prototype in two categorizes.

1. One is the performance problem.
2. The limitations the prototype comes with such as (section 6.4 on page 34):
    - file owners
    - file permissions
    - file format between different servers

Solving these set of problems would allow us to provide a usable replication feature. The most important of such limitations can be eliminated by modifying the *Capture* code to understand a Data-Link datatype. Once the *Capture* code knows the Data-Link datatype setting up the replication environment should not be a problem anymore.
The other major limitations can be addressed by making the Stored Procedure more dynamic, leaving all the hard coded parts out of it since it is not possible to pass any parameters to a Stored Procedure which is called by a subscription. This would have to be dealt with first. The performance problem can be addressed by eliminating the BLOB copies. This can be achieved by using a regular FTP tool to copy the files and restrict the use of *Capture*

and *Apply* to replicate the file references.
Once the above points are done, the customer would have a usable replication product for data linked files. This is the current approach planned for inclusion in IBM's DPropR product.

References

[1] T.Beuter, P.Dadam: Prinizipien der Replikationskontrolle in verteilten Datenbanken; *published in: GI Informatik und Forschung und Entwicklung, 11, 4, (Nov. 1996), 203-212*

[2] Pam Drew, Roger King, Dennis McLeod, Marek Rusinkiewicz and Avi Siberschatz. "Report of the Workshop on Semantic Heterogeneity and Interoperation of Multidatabase Systems." SIGMOD Record, Volume 22, Number 3, September 1993.

[3] Robert Goldring: Update Replication: What Every Designer Should Know. Info DB (USA) Vol.9, No.2 April 1995, *published at: http://www.software.ibm.com/data/pubs/goldring/*

[4] Andreas Heuer, Gunter Saake: Datenbanken Implementierungstechniken; *1. Auflage 1999*

[5] Judith R. Davis, DATALINKS: MANAGING EXTERNAL DATA WITH DB2 UNIVERSAL DATABASE Prepared for IBM Corporation by February, 1999 *published at: http://www-4.ibm.com/software/data/pubs/papers/datalink.html*

[6] IBM bookserver, Title: DB2 Replication Guide and Reference Document Number: SC26-9642-00, Build Date: 03/02/99 14:03:22 Build Version: 1.3.0 Book Path: /home/publib/epubs/book/ASNU5002.BOO *published at: http://publib.boulder.ibm.com:80/cgi-bin/bookmgr/-BOOKS/ASNU5002/CCONTENTS*