UNIVERSITÀ DI PISA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

A CONSISTENCY SERVICE FOR REPLICATED HETEROGENEOUS DATABASES IN A GRID ENVIRONMENT

Relatore:
Prof. ANDREA DOMENICI

Relatore:
Prof. GIANLUCA DINI

Candidato:
LAURA IANNONE

ANNO ACCADEMICO 2004-2005

Abstract

The present work, written during a period of scientific collaboration with INFN[1] (National Institute of Nuclear Physics), proposes a mechanism for maintaining consistency among replicated heterogeneous databases. This mechanism is integrated in a Replica Consistency Service for Data Grid environments.

Nowadays, an increasing number of applications, manage an enormous quantity of data distributed on a very large scale. In a Data Grid the data files are replicated to different sites. The replication of data among multiple databases is the only way to ensure that the data are available where and when they are needed.

The catalogs, used for maintaining information about the state of file replicas in a Data Grid, use replication to improve availability, to reduce data access latency, and to improve the performances of distributed applications. Moreover it is useful for fault tolerance.

A Data Grid may serve different user communities (Virtual Organizations), with different requirements. Therefore different types of data may be stored in different formats and in storage devices of different vendors. A mechanism is needed to maintain consistency among the replicas of a certain catalog that uses heterogeneous back-end databases.

In a Data Grid, heterogeneous data storage involves the problem of a uniform interface to access heterogeneous information; this problem requires tools for data integration.

Our goal for this work is to provide the ability to manage and to execute a replica synchronization; in particular we have a replica on an Oracle server and another replica on a MySQL server with the same logical schema. With the word "synchronization" we mean that an update of the target database will be done. Only the changes that happen on the source database will be propagated to the target database.

Acknowledgments

I would like to thank all the people who have supported me during this work.
For their constant and precious support I want to thank Flavia Donno (INFN Pisa) and Dirk Düllman (CERN).
I want to thank also my supervisor Andrea Domenici (University of Pisa) for introducing me to Grid Computing.
For their collaboration many thanks go to Gianni Pucciani (INFN Pisa) and Heinz Stockinger (University of Vienna).
Thank you to Barbara Martelli (CNAF), Elisabetta Vilucchi (CNAF), Eva Dafonte Perez (CERN) for their help and assistance.

Contents

List of Figures

Chapter 1

Grid Data Management

This chapter introduces the concept of *Grids* and their developments. In particular, Data Management issues are discussed.

1.1 Grid Networking

Grid Computing [2, 3] is an important new field of research for Information Technologies.

The Grid is an infrastructure for sharing computer power and data storage capacity of possibly millions of systems across a worldwide network. It links together huge computing resources and provides mechanisms to access them easily from different platforms. The evolution of high-speed data networking technology has allowed the development of Grid computing.

Grids enable the creation of *Virtual Organizations* (V.O.), sets of users that put together and share their resources in order to achieve a common goal. Grid technologies allow an easy, coordinated and secure way to access these resources for a better utilization.

Enterprises and istitutions (in such fields as life sciences, energy and oil, manufacturing, financial, government) are beginning to show interest in Grid computing. A growth of the Grid computing market, above all in education and research, is expected in the next years.
There is, therefore, a need of optimal solutions for solving Grid-related problems.

1.2 Grid relationships with other technologies

The essence of the definition of Grid, as written in [4], can be given by the following list, according to which a Grid is a system that:

1. "coordinates resources that are not subject to centralized control", Grid resources and Grid users can live within different control domains;

2. "using standard, open, general-purpose protocols and interfaces", multi-purpose protocol and interfaces must address the issues of authentication, authorization, resource discovery and resource access in a Grid environment;

3. "to deliver nontrivial qualities of service", Grid resources must be used so that quality of service of various level is guaranteed.

According to this list **cluster systems**, that can be defined as a collections of homogeneous servers aggregated for increased performance, are not Grids, because they have centralized control of the hosts that they manage. They have complete control of system state and user requests.

Trying to find another candidate that satisfies the three previous points we can consider that the **Web** is not a Grid, despite its protocols support access to distributed resources; in fact these resources are not used in a coordinated fashion to reach a satisfying quality of service.

Distributed computing systems, **peer-to-peer systems** and **multi-site schedulers** (such as, e.g. *Platform's MultiCluster*) can be considered as a first generation of Grids. They manage distributed resources (but in a narrow domain) with interesting qualities of service, without a centralized control.

Then, current distributed computing systems do not provide a generic approach to resource sharing, as required by *Virtual Organizations*.

The three points can be clearly applied at all the system developed in projects of scientific communities, as *Data Grid project*, that integrate resources from different institutions, each with own policies and mechanisms of resources management. These systems address various qualities of service, regarding for example response time, availability and security and provide common standards for services.

1.3 Types of Grids

The concept of Grid is based on the sharing of resources; there are different typologies of resources, then different type of Grids may be characterized accordingly. These Grids may have differerent architectures.

A **computing Grid** is a collection of distributed computing resources that can be either within or between administrative domains.
This aggregation acts as a virtual supercomputer. It performs sequential or parallel jobs efficiently and provides computational power on-demand, independently of the machines that provide it.

A **data Grid** provides secure access to an enormous quantity of data distributed in different domains. Today data Grids are being built to serve research communities and several software vendors provide data Grid solution for business applications.

1.4 Grid architecture

The layers that define a Grid architecture [5], from top to bottom, are:

- Application layer
- Collective layer
- Resource layer
- Connectivity layer
- Fabric layer

Each layer provides a specific function. The higher layers are more software-centric, whereas the lower layers are more hardware-centric.

The **Application layer** is the highest layer of the Grid. It includes all different user applications, portals and development toolkits supporting the applications.

The **Collective layer** provides services that are not associated with any one specific resource but are global and are used to coordinate multiple resources (e.g. resource discovery, resource brokering, system monitoring, community authorization, certificate revocation). In this layer is implemented the **Data Replication Service**, that allows the management of resources to maximize data access performance

The **Resource layer** is built on the underlying protocols to define protocols for access and management of the local resources. Resource layer protocols can be used to obtain information about the resources and their state (*Information protocols*) and to access to a shared resource to perform an operation (*Management protocols*).

The **Connectivity layer** provides communication protocol to enable the exchange of data between Fabric layer resources, and authentication protocols to verify identity of users and resources. Often standard protocols, developed within the context of the Internet protocol for communicating easily and securely, are used.

The **Fabric layer** provides the resources whose shared access is mediated by protocols. A resource can be a hardware element (e.g. storage system, CPU, catalogs) or a logical entity (e.g. a distributed fyle system). This layer implement the local operations that regard specific resources as a result of sharing operations at higher levels.

This multilayer architecture requires standardization. There are several projects that support the development of Grid standard: *Global Grid Forum (GGF)*, *Globus*, *Open Grid Service Architecture (OGSA)*, and others.

1.5 Data Management in Grid computing

1.5.1 Data Management overview

An incresasing number of research projects currently need tools and infrastructures to produce, analyze and transfer information in quantities in the order of Petabytes. So they require a data management system [6, 7] that enables a secure and fast access to these data, moves and replicates them between geographically distributed locations at high speed, and manages synchronization of remote copies. Data caching

and replication are used to minimize network traffic.
This system must guarantee fast transfer of files between heterogeneous mass storage systems over the Grid.

The solutions provided have to ensure easy use, scalability and security to manage the data sources.
These distributed data sources may be files and databases; they may be different in their formats, access mechanism, ownership, access policies.
The Grid must be able to provide data virtualization for a transparent data access and processing; all the resources of the same kind should be accessed in a uniform way, independently from the technologies or standards they are based on. Resource heterogeneity should be hidden and a uniform interface should be provided. Virtualization helps to produce a data management solution with much more flexibility than traditional approaches to data access and storage.

The data management requirements can be related to the following areas:

- **Data discovery and access**. This area includes the ability to discover the data through metadata examination. Once a set of relevant data has been identified, the next step is the access to the data.
  In data transfer, high transport performances are required and parallelism is exploited.

- **Data exploration and analysis**. The accessible data can be processed in various ways (e.g. encrypted, decrypted, decompressed, compressed) before being delivered to user applications. They can be combined with local data or with other remote data. The transformations and the analysis performed require complex mechanisms to manage the data.

- **Resource management, security and policy**. Managing Petabyte volumes of data requires efficient and coordinated control of a huge number of network and computational resources at multiple sites. In particular, resource management must deal with equipment failures.
  Finally a secure data access mechanism and high-throughput computation must be guaranteed.

1.5.2 Data Management Components

A data management service [8] consists of the following components:

- **Replica Manager**, it manages files and meta data in a distributed and hierarchical data storage model (as that developed by the MONARC project [9]);

- **Data Mover**, it is used by the Replica Manager to transfer files from one site to another;

- **Data Accessor**, it is an interface for the local file system selected by the Replica Manager;

- **Data Locator**, it is used, with Data Accessor, by the Data Mover to implement its functionality; it is responsible for accessing physical files;
- **Meta Data Manager**, it manages a distributed and hierarchical set of objects;
- **Query Optimization**, it provides an optimal migration and replication execution plan for a query.

Each component ensures security during its activity.

1.5.3 Distributed Database Management Systems

Originally, in the early Grid projects, the data are stored mostly in files; currently also object-oriented and relational databases are used for data storage. This requires that **database management systems** (DBMS) [10] must be integrated into the Grid.
The existing DBMS have been made for commercial environments that do not have very high scalability requirements, unlike those in Grid environments. Then, they must be enhanced.

In a DBMS there is a single access method, independently of the amount of data accessed and of the type of operations performed. For a data Grid a single access method is not a good choice, because it depends on the number of objects that must be accessed from a file. In fact, to access a large amount of data of a remote file, it is more efficient in term of response time to use a file transfer mechanism to transport the entire file, rather than access remotly single objects of the file
In a DBMS it is useful to identify a standard unit of data access and transfer based on the types of services that use the stored data.

In a data Grid the data files are replicated to different sites. Replication reduces data access latency and improves performances of distributed applications. Moreover it is useful for fault tolerance.

In the previous sections we have addressed data replication of generic files. In this section we add some details about database replication.

The applications do not need to know where the data is located. The dbms provides, then, the management of the replicas[1] of the data. Metadata structures are necessary to store information about the distribution of data; they must be replicated too.
Data replication introduces problems, such as consistency and allocation of the replicas; update and write operations on a replica need to be synchronised with others replicas.

A Data Grid may serve different user communities (V.O.), with different requirements. Therefore different types of data may be stored in different format and in storage devices of different vendors. DBMS must support heterogeneous data storage.

[1] A physical instance.

The problem of an uniform interface to accessing heterogeneous information requires tools for data integration.

The DBMS provides two kinds of virtualizations to the users:

- *heterogeneity transparency*, to hide the data formats, the data access mode, how the data are stored and managed;
- *distribution transparency*, to hide the fact that the data sources are distributed on a wide area, and then a network communication is needed to access them.

When a user submits a query to the DBMS, it must provide an optimal execution plan for identifying and obtaining the requested data.

For ensuring the security of data, DBMS should provide mechanisms to perform authentication of the end users, and once a user is authenticated, it must determine what data he is authorized to access.

Chapter 2

Replica consistency in a Data Grid

This chapter illustrates the problem of replica synchronization and consistency in a Data Grid.

2.1 Data consistency

Nowadays Data Grids are a possible solution to the problems of management of big volumes of data on a world-wide distribution. In a typical Data Grid there are petabytes of data replicated all around the world. A replica is not just a simple copy of a file; replicas are two or more physical instances of the same logical file and some metadata information about replica locations. Update operation on one replica must be propagated to the other replicas.

Then, one of the most important challenges in Data Grid management is to provide consistency at the different levels of the Grid [10, 11].

If in a Grid the data are read-only, they are always consistent because no updates on the replica are done. The only requirement is that these files are created and have a unique names. It is a low consistency level problem. If updates are possible on the replicas, the consistency problem must be addressed.

It is also important to have a policy that specifies who can make the data changes. We can have a **well defined file owner**, that is the only entity that can make the changes on the data. In this case the replication is not a symmetric process; to obtain the updated version of data it is necessary that the ownership information are propagated to the replica. Write and update operations must be forwarded to the master replica; read access, instead, can be forwarded to any replica. We can also have **varying ownership**; in this case there is need to contact all replicas for obtaining a quorum, before doing an update operation.

The degree of data constistency depends on the update frequency, amount of data items covered by the update and the expected response time of the replicated system.

In a Data Grid we have different types of data, that require different consistency levels. Then a data consistency service, for a Grid, should offer several replication policies. Moreover, consistency models and tools must be applicable to a large variety of heterogeneous data store types.

In a Data Grid the *Replica Manager* allows users to create, move, update and delete replicas. This means that files are copied from one site to another and registered in a catalogue. Additional information, such indexes and catalogue, used to manage the data, can also be replicated and then consistency problem also concern them.

2.1.1 Synchronous model

Synchronous model requires that update operations on all the replicas are done within a single transaction[1].
If we succeed in establishing a synchronous replication, each local database transaction is propagated to all the other replicas of the data involved in the transaction, or at least at the majority of replicas.

In a DBMS this synchronous transaction can be established by extending the locking mechanism to all the replicas. But if we use a distributed DBMS we not provide a flexible solution, becouse we have not different cosistency levels for different kind of data; we can have some types of data that can be out of synchronization (low consistency) and other types that always need to be synchronised (high consistency).

2.1.2 Asynchronous model

Keeping petabytes amount of data synchronized is not a very good solution for a Grid environment.
The method of asynchronous replication allows some replicas, for a given amount of time, to be in an inconsistent state.
Common asynchronous solutions are:

- **Primary-copy approach**, for each data item there exists a primary copy and all the other replicas are secondary copies. The changes can be made only to the primary copy; update requests sent to a secondary copy are passed to the primary copy which does the update and propagates the changes to all the secondary copies.

- **Epidemic approach**, changes may be performed on any single replica; a separate activity compares version information of all the replicas and propagates the updates to older replicas in a lazy way.

- **Subscription and relatively independent sites**, a site does not care about consistency. When a replica is updated, the changes are notified only at certain

[1] A transaction is an atomic unit of database access which is either completely executed or not executed at all.

sites. A site that does not receive the changes must get the latest information from other sites.

2.2 Replica Consistency Service

The **Replica Consistency Service** (RCS) is a Grid service that must guarantee the consistency of replicated data.
This service can take into account different replication schemas and can guarantee different consistency levels for user applications.

Data consistecy levels, as discussed in [11], are:

- **possibly inconsistent copy**, the file replica is created using a trivial file copy concurrently with ongoing write operations. Then, while a user updates a file, another user can copy the file to another location. The data may be inconsistent.

  There are several mechanisms, such as obtaining a file write lock before performing the file copy, that solve this problem and coordinate the data access between local users and replication system.

- **consistent file copy**, the replica corresponds to a snapshot of the original file at a certain point. If the file is controlled by a DBMS, the snapshot can be taken in the middle of a transaction. Therefore we can have a dirty read problem, because a reader accesses a file while another user writes to the same file.

  In this case to maintain the consistence, when we would like to get a replica, we can use a read lock for excluding other writers or a MROW (multiple-reader-one-writer) read lock to allow a concurrent user to make a write operation.

- **consistent transactional copy**, to obtain a replica no write transaction must be active. A consistency problem could take place when a job requires two files and a data object in a file contains references to objects in the other file that is deleted or updated between the two copy operations. Possible solutions to eliminate inconsistency are to produce all replica files as part of a single database transaction or to remove cross file references. This level is the minimal consistency provided by a DBMS that uses transactions.

- **consistent set of transactional copies**, at this level a local DBMS manages one instance of a physical file and a replica of this file is stored at different site and managed by a different DBMS. Then, replica and original file are free to diverge.

- **consistent set of up-to-date transactional copies**, also called synchronous replication. A replica is under the control of a DBMS; read or write accesses to a replica are atomic transaction and the locks are negotiated over a WAN.

An important difference between a Replica Consistency Service for a Data Grid and a distributed DBMS are that the first manages heterogeneous data, whereas the second control homogeneous data. Moreover the RCS handles a large amount of data, and then needs scalability.

2.2.1 RCS architecture

A basic architecture for an RCS, as described in [12], must provide the following functionalities:

- client interface for the service;
- file update mechanism, that is responsible of applying changes to a single replica;
- update propagation protocol, that is responsible of propagating the update at all the replicas over wide area networks.

In a generic Data Grid there are two kinds of Grid nodes: *Computing Element* that provide computing power, and *Storage Element* (SE), a basic storage resource in a Data Grid. The interface to a SE must be unique, independently of the underlying storage technology.

The RCS components are, as shown in figure 2.1[2], the following:

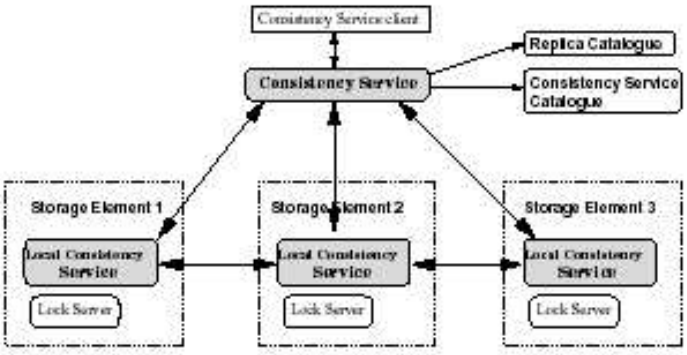

Figure 2.1: Architecture of the consistency service

- **Consistency Service**, provides the main interface for an user through a consistency service client. It makes available the interface to update distributed data stores, with the aid of the other components of the architecture.
- **Transaction System**, for the update propagation protocol as well as for all communications between components.

[2] from [12].

- **Local Consistency Service** (LCS), allows a local update and propagates the changes using the Transaction system.
- **File Lock Service**, it is responsible for holding and releasing locks for all files recorded on a storage element.
- **Replica Catalogue**, to retrieve information about the replica locations. It provides mappings between logical file names and the physical storage system locations of one or more replicas of these files.
- **Replica Consistency Catalogue**, manages all the necessary information for the file update and update propagation.
- **Replica Manager**, transfers file updates to remote sites using the Replica Catalogue service.

2.3 Use Cases

To illustrate consistency problems in an heterogeneous database synchronization environment, let us consider some common use case scenario.

2.3.1 VOMS

VOMS [13, 14, 15] is the **Virtual Organization Membership Service** that manages the authorization information for a VO. The purpose of the VOMS is to manage authorization data about users of a VO.

Each VO has its own VOMS. Users of a VO are organized in groups forming a hierarchical structure with the VO as the root. Users are characterized by roles that they cover in a group and by capabilities, properties to be interpreted by the local sites.

Every user in a VO is characterized by a set of attributes: group, role and capability. Combinations of values of these form unique attributes called **Fully Qualified Attribute Names** (FQAN). A FQAN has the following form:

/VO/[group[/subgroup(s)]][/Role=role][/Capability=cap]

The VOMS server is essentialy a front-end to an RDBMS (MySQL or Oracle), where all the information about users is kept.

Each VO maintains a VOMS database. Tables of the VOMS database schema can be:

- *Data tables*, contain the actual data that define the VO.
- *Archive tables*, contain old data that was deleted or changed in data tables.
- *Service tables*, contains information useful for timekeeping, sequence generation, request handling.

Each VO can have different tables in the VOMS database.

The VOMS system is composed by the following parts:

- **User Server**, returns authorization info to the client. There is one instance of the server for each VO.
- **User Client**, presents a user's certificate to the server and obtains a list of groups, roles and capabilities of the user.
- **Administration Client**, used by the VO administrators for adding users, managing groups, etc... .
- **Administration server**, accepts requests from the clients and update the VOMS database.

All client-server communications are secured and authenticated. In a VOMS session:

1. the first operation is a mutual authentication between client and server;
2. then, a client sends a request to server;
3. the server checks that the request is correct;
4. server sends back the required information;
5. client checks the validity of the info received;
6. client repeats steps 1-5 for each server that it needs to contact.

A VOMS server can not be centralized, it needs replication for guaranteeing fault tolerance and load balancing. The master can be Oracle or MySQL and its update frequency is rather low.

2.3.2 RLS

Another typical use case is the **LCG Replica Location Service** (LCG-RLS) [16], a central file catalog for the LHC Computing Grid [17]. It mantains a consistent list of accessible files together with their relevant file metadata attributes. When a file is created, the RLS associates a unique and immutable identifier with the file.

RLS uses Oracle Application Server and Database. It is currently centralized with a single database backend. A single central catalog is a problem for scalability and availability. For this reason, RLS needs consistent replication among multiple catalog servers. The number of foreseen replicas is about 5 to 10. For RLS the update frequency is higher, but an update delay of 1 hour is tolerated.

Chapter 3

Constanza: a Replica Consistency Service for Data Grids

In this chapter we present a Replica Consistency Service (RCS) prototype, developed in the *INFNGrid* project [18], that enables users to achieve consistency among replicas in a Data Grid environment.

3.1 Overview

Constanza [19, 12] is a RCS accessible as a Web Service. It can be configured for different update propagation policies; currently *Constanza* allows replica updates in a single-master scenario with lazy update synchronization[1]. Replicas can be relational databases or flat files.

3.2 Architecture

The main components of the *Constanza* architecture are the following:

- **Global Replica Consistency Service (GRCS)** main component and interface to the end users.
- **Local Replica Consistency Service (LRCS)**, allows replica updates; internally it has a catalogue to manage specific metadata that are required to update the local replica. In *Constanza* a single-master policy is implemented: a single LRCS holds the master replica that can be updated by the end users and all others replicas are synchronized by the RCS.

[1] Each replica is updated in a separate transaction.

- **Replica Consistency Catalogue (RCC)**, it is responsible for storing the metadata information, such as replica location and replica status.

In figure 3.1 is shown the interaction among these components. The interface to the

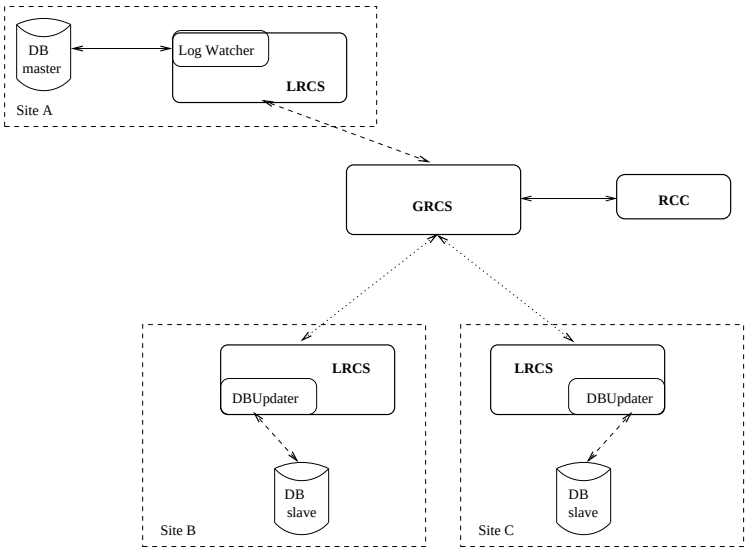

Figure 3.1: Architecture of the *Constanza* service

RCS is a command line client that sends update requests. An user first updates the master replica and then issues the update propagation command.

In a database replication scenario, a **Log Watcher** component is added to this architecture. Periodically it checks if updates have been made on the master replica. When this happens it extracts the last updates from the database log files and puts them in an update file that will be sent to the others replicas with the `update` command.

The changes will be applied at a secondary replica by the **DBUpdater** component.

Constanza supports an asynchronous propagation policy, that violates the consistency requirement for the time necessary to complete the update propagation.

3.3 Implementation

A *Constanza* prototype has been implemented in C++. The main subsystems of the service are shown in figure 3.2.
The **Core** package implements the subsytem functionalities.
The **Comm** package allows communication among subsystems and between the service and the clients.
The **DSInterf** package it is responsible for applying updates. For flat files this module can work with different Storage Element interfaces. For databases this

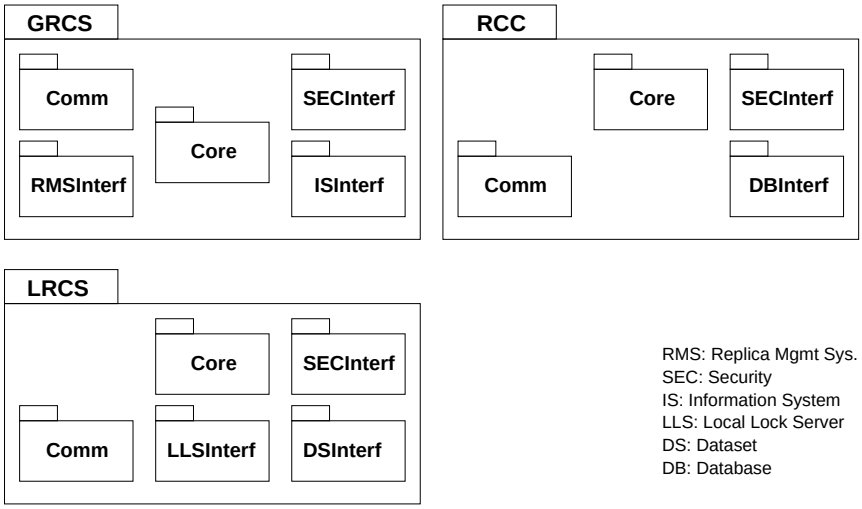

Figure 3.2: Subsystems of the RCS

module must be able to provide a heterogeneous database synchronization. RCS sends a log file to the LRCSs responsible for the database replica; DSInterf applies the update through a **DBUpdater** module, that translates the updates in the format required by the local database.

3.4 Future Work

Constanza currently provides log extraction and application only between MySQL databases.

This thesis aims to provide a solution that extracts update information from an Oracle log file and then imports and applies these updates to a MySQL DB.
With the purpose to extend *Constanza* to an Oracle-MySQL replication, we have examined a few existing off-the-shelf tools and some utilities provide by the Oracle server.

Chapter 4

Off-the-shelf tools for heterogeneous databases synchronization

This chapter provides information about data replication tools available off-the-shelf. We will look in detail into two products that can solve the problem of maintaining consistency among heterogeneous replicas: **Enhydra Octopus** and **DBMoto**. In particular, Oracle to Mysql replication is used to evaluate these applications.

4.1 Enhydra Octopus-3.2.2

Enhydra Octopus [20] is provided by *ObjectWeb Consortium*, relased under *LGPL*[1].

This application is a simple Java-based extraction, transformation and loading tool.
It loads data from a JDBC data source (database) into a JDBC data target (database), and it performs many transformations defined in an XML file (figure 4.1[2]).

Octopus supports a wide range of JDBC data source types, which can be used both as source and target of the data transfers: CVS, XML, Excel, Access, MSSQL, MySQL, InstantDB, PostgreSQL, QED, HSQL, Oracle, DB2, McKoi, Sybase, Paradox and Borland JDataStore.
Octopus scripts are fully portable in terms of operating system.
The Octopus replicator works in two steps:

1. generation of database schema description and conversion scripts;
2. actual database replication.

The Enhydra Octopus architecture consists of two parts:

[1] Lesser GNU General Public License.
[2] From the Enhydra Octopus Application manual.

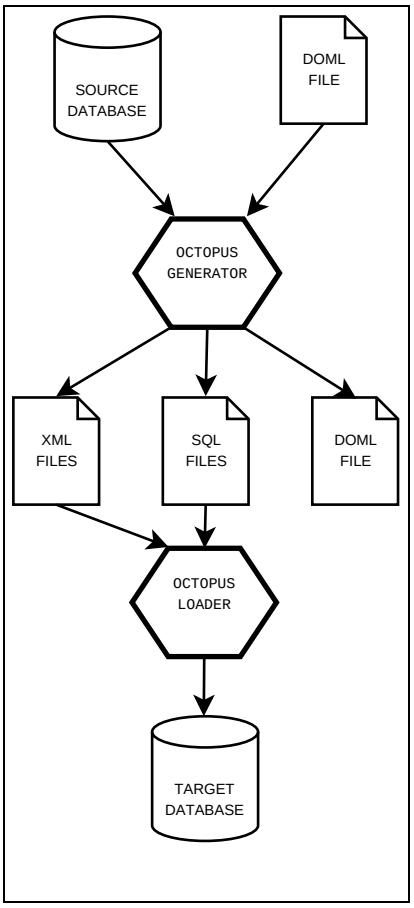

Figure 4.1: Enhydra Octopus architecture

- **OctopusGenerator**
- **OctopusLoader**

OctopusGenerator and OctopusLoader share the same GUI.
The application requires Java 2 SDK Standard Edition (version 1.4.x or grater).

4.1.1 Enhydra Octopus Generator

OctopusGenerator should be run when the database schema changes.
It reads existing data structures and generates corresponding XML files (loadJob.xml and importDefinition.xml) automatically.

A user can choose to generate SQL and doml[3] files too.
SQL files include sql statements for creating database, tables, primary keys, indexes and foreign keys.

[3]DODS (Data ObjectDesign Studio) file, which is XSL based tool for communication with databases.

To start OctopusGenerator you must set some parameters:

- source database type (e.g. MSQL, Oracle, DB2, MySQL, etc...);
- target database type (e.g. MSQL, Oracle, DB2, MySQL, etc...);
- source database URL (e.g. localhost:3306/RCC);
- target database URL;
- Doml URL, when use doml file as input;
- source database driver name;
- target database driver name;
- source database user;
- source database password;
- target database user;
- target database password;
- value mode, that is the difference between overwrite and update attributes. Possible values should be Copy and Sync, but only Copy mode is avaible in this version, Sync mode is not implemented;
- generator output directory, where OctopusGenerator places all created files;
- include table list, tables that are included into Generator process;
- additional classpaths, path for the .jar files.

You can also indicate if Generator must create sql files as output, and in this case which sql file will be generated. You can choose to generate xml files and doml files as the generator's output.

4.1.2 Enhydra Octopus Loader

OctopusLoader should be run when data change. It may be connected to any JDBC data source and performs transformations defined in an XML file.

Actually it performs an overwrite of the target table, not a synchronization[4]. In order to do that, OctopusLoader needs XML and SQL files created by OctopusGenerator and it must be connected with the source database.

If you want to start OctopusLoader you must set the following parameters:

- log mode;

[4]With "synchronization" we mean an update of the target table will be done. Only the changes that happen on the source table will be propagated to the target table.

- restart indicator, the possibility to continue the process where it was stopped without losing or duplicating data, if it was stopped;

- log file directory, where OctopusLoader places all created files;

- log file name;

- additional classpath, path for the .jar files.

4.1.3 Enhydra Octopus In Atlas

Enhydra Octopus is used in the *Atlas HEP experiment*[5] to replicate databases from Oracle to Mysql.

In the Atlas application some patches to Enhydra Octopus have been introduced; one patch converts the Oracle/varchar2 type to Mysql/varchar type, another patch creates commands which accept `PRIMARY` as a Primary Index name in the Mysql tables.

4.1.4 Evaluation

To test the product we have used an *Oracle10g* database as a JDBC data source and a *Mysql 4.1.9* as a JDBC data target. The same table has been created in both databases.

We have started the Octopus Generator via GUI (graphic user interface) and from the 'Applications' menu we have selected 'New Octopus Generator'. After that, a user frame is shown on the screen.
We have entered the following parameters into the frame fields that are in the 'JDBC' tab (figure 4.2) :

- source database type: `Oracle`;

- target database type: `MySQL`;

- source database URL: `@oracle-db-3.cr.cnaf.infn.it:1521:pisatest`;

- target database URL: `localhost:3306/RCC1`;

- source database driver name: `oracle`;

- target database driver name: `mm`;

- source database user;

- source database password;

[5] ATLAS (`http://atlas.web.cern.ch/Atlas/`) is a particle physics experiment that will explore the fundamental nature of matter and the basic forces that shape our universe. The ATLAS detector will search for new discoveries in the head-on collisions of protons of extraordinarily high energy.

- target database user;
- target database password;
- value mode: `copy`;
- generator output directory;
- include table list: `GRCSs`;
- additional classpaths.

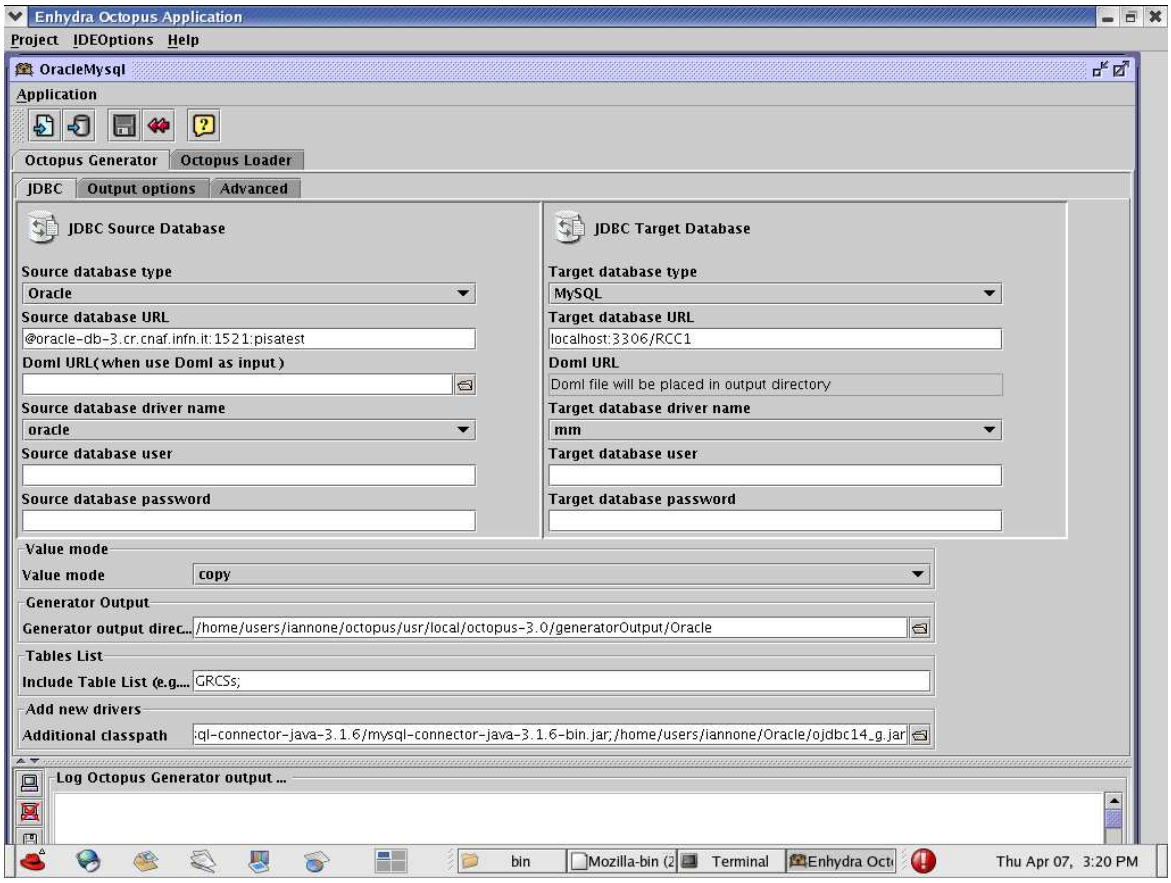

Figure 4.2: Enhydra Octopus Generator

In the 'Output options' tab (figure 4.3) we have checked the "Generate Xml files" options and the "Full mode for all tables" option.

On the 'Advanced' tab (figure 4.4) we did not check any option.

After entering all needed parameters, we have pressed the "start" button.

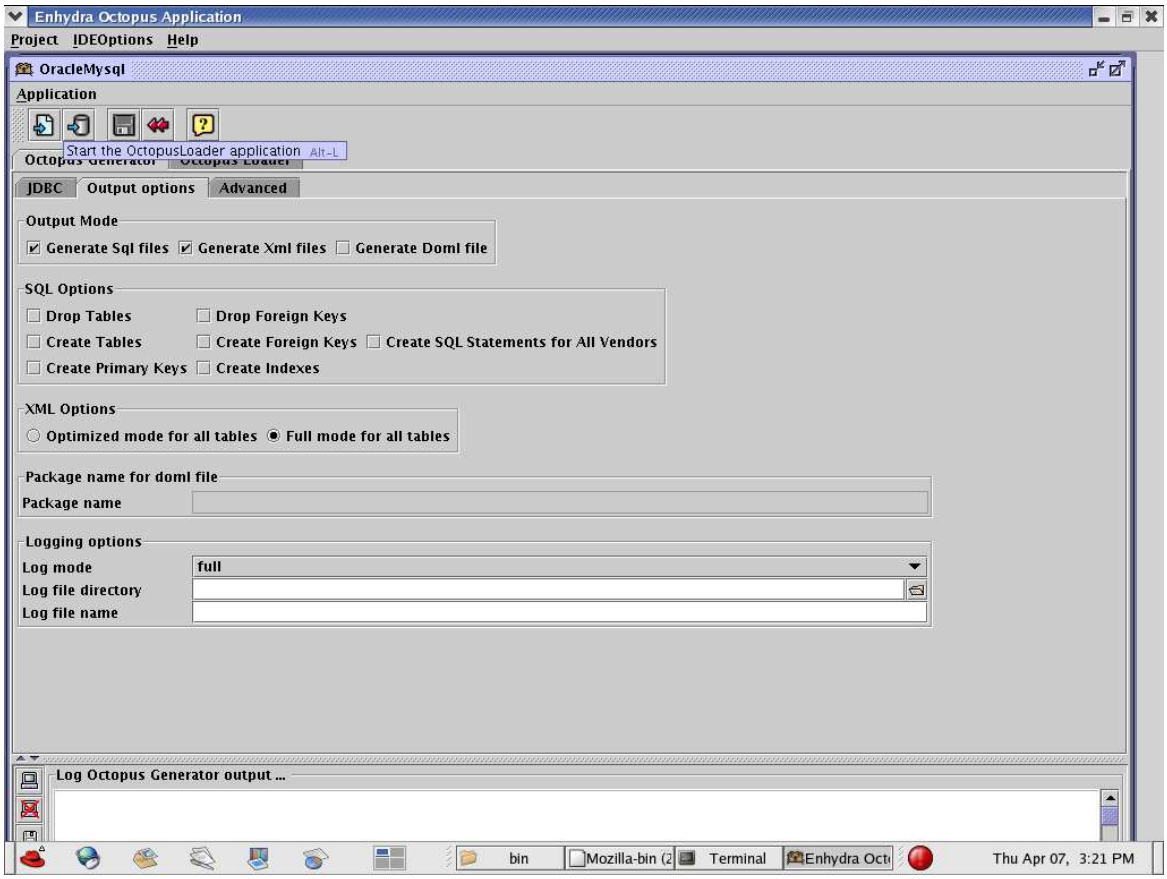

Figure 4.3: Enhydra Octopus Generator

Once the Octopus Generator is connected to the source database, it reads all meta data which describe relationships between tables and columns within tables; these relationships are stored in appropriate files that Octopus Generator creates.

In our test Octopus Generator has created the *LoaderJob.olj* file and the *ImportDefinition.oli* file.

The first one stores information about JDBC parameters for source and target database, and information about the generation of SQL files and data import options (path to *ImportDefinition.oli*). Our *LoaderJob.olj* is shown in appendix A.1.

The *ImportDefinition.oli* file stores information about source and target tables, needed to copy source database to target database with Octopus Loader application. Our *ImportDefinition.oli* is shown in appendix A.2.

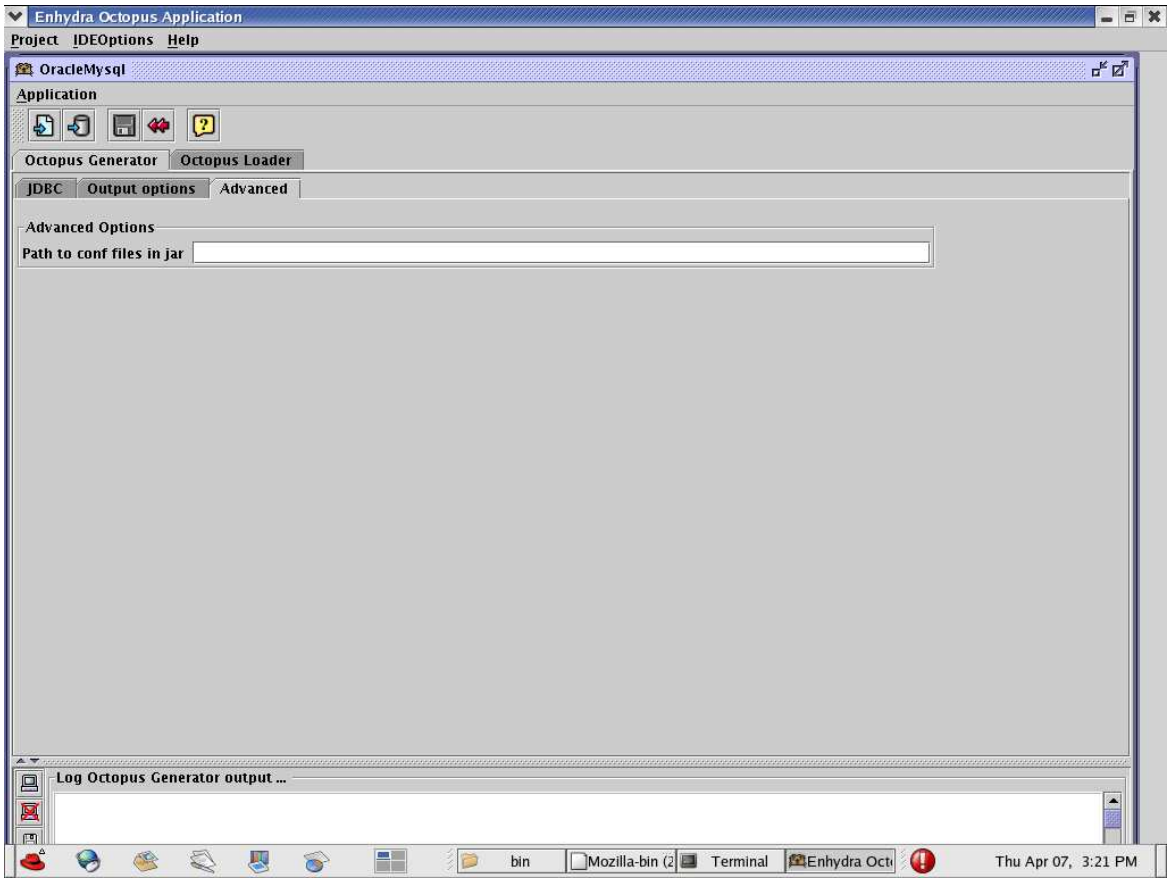

Figure 4.4: Enhydra Octopus Generator

Subsequently we have selected "New Octopus Loader" from the menu' "Applications", and then we have entered the following parameters into the frame field (figure 4.5) :

- log mode;
- log file directory;
- on error continue: `true`;
- path to loaderjob.olj file;
- additional classpath.

After this, we have pressed the "Start" button: Octopus Loader connects itself to the source database and performs the transformations defined in the XML file.
The Octopus Loader output is shown in appendix A.3.

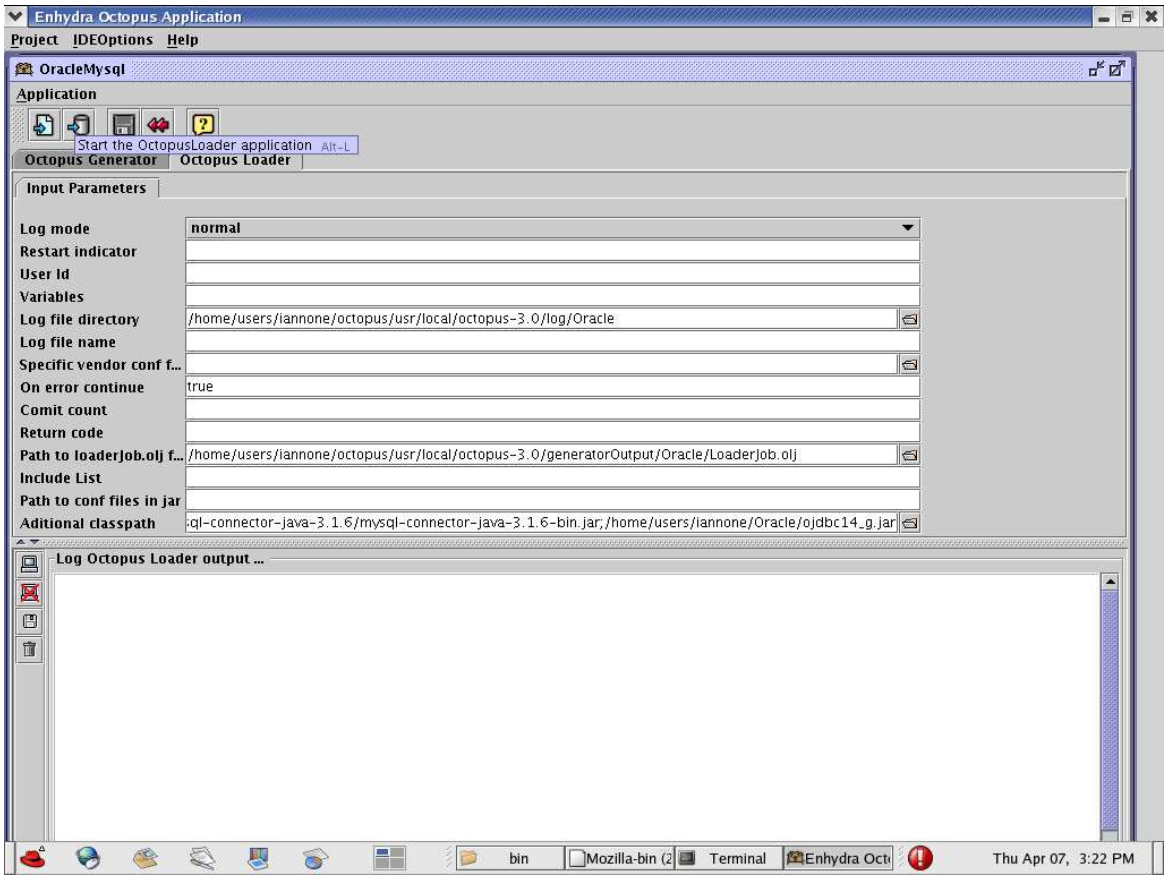

Figure 4.5: Enhydra Octopus Loader

Update of the target database is done by overwriting the rows of the target table. If you delete a row from the source table, this row remains in the target table until you drop the target table and create it again.

Our test is mainly about database synchronization: we have considered the MySQL database as a replica of the Oracle database that must be refreshed. But we have also tested the creation of the target table in the target database[6]. In this case in the generated files there are the following SQL files too: *CreateTables.sql*, *CreatePrimary.sql*.

We have checked that Octopus Loader works properly and generates an output file shown in appendix A.4.

[6]We have checked the "Generate Sql files", "Create tables" and "Create Primary Keys" options in the 'Output options' tab

4.1.5 Comments

We can say that Enhydra Octopus, currently, provides an Oracle to Mysql database replication but not a synchronization. It has, however, some advantages as an open source product already used and tested in the *ATLAS* experiment.
Enhydra Octopus supports a wide range of database vendors and it is simple to use.

4.2 DBMoto

DBMoto is a user friendly product, developed by HITSoftware [21].

It performs data replication between a source database and a target database. (figure 4.6). DBMoto can manage one or more source databases and one or more

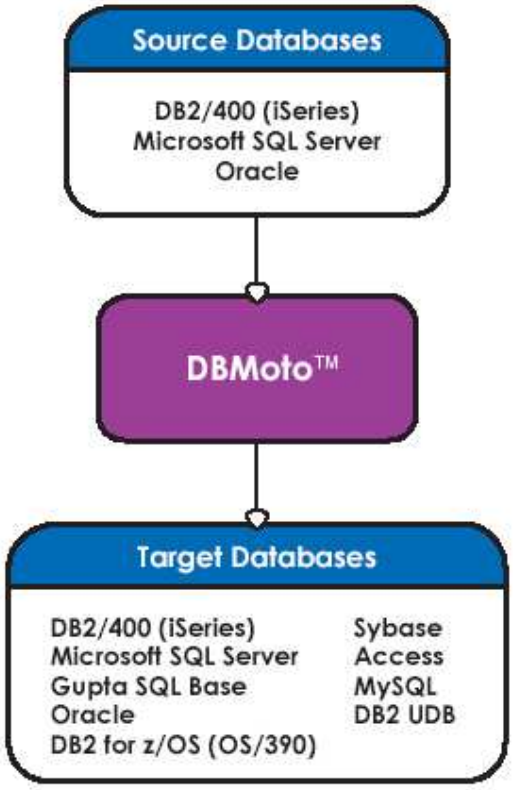

Figure 4.6: DBMoto

target databases.
DBMoto logs all the information about the replication process in a database called Metadata database (using any database among Sybase, MySQL, DB2/UDB, MS Access, DB2/400, Microsoft SQL server and Oracle).

DBMoto runs on Windows: the entire replication process is configured and managed from a Windows platform.

DBMoto is divided into two componets: **Enterprise Manager** and **Data Replicator**. The first is an interface that allows users to:

- configure source, target and metadata database;
- view logs;
- create target tables;
- define replication criteria.

It includes Wizards that help in the setup replication process.
Data Replicator manages the data replication process.

The Metadata, Source and Target Connection Wizards simplify the work of configuring the environment for replication.

DBMoto has three operation modes:

- *Refresh*
- *Mirroring*
- *Synchronization*

In *Refresh mode*, DBMoto reads a large amount of data from the source database, applies administrator-defined mapping rules, and writes the result on the target database.

In *Mirroring mode*, DBMoto performs a real time incremental replication based on log management.

Synchronization mode is similar to mirroring, but in bi-directional replication fashion[7].

From DBMoto you can also visualize all the rows of the source and target table.

4.2.1 Evaluation

For evaluating the product we have used the following configuration:

- DBMoto version = 4.01.008, a Metadata database type = Access, an Oracle10g client, and a Mysql client on WindowsXP platform;
- Source database type = Oracle 10.1.0.2.0, on Scientific Linux platform;
- Target database type = MySQL 4.0.22, on a Red Hat Linux platform;

[7]Each machine is both source and target.

Before starting DBMoto, it is necessary to create all the ODBC or OLEDB connections required for correct working (figure 4.7).

In our case we have used "Microsoft OLE DB Provider for ODBC Drivers" for

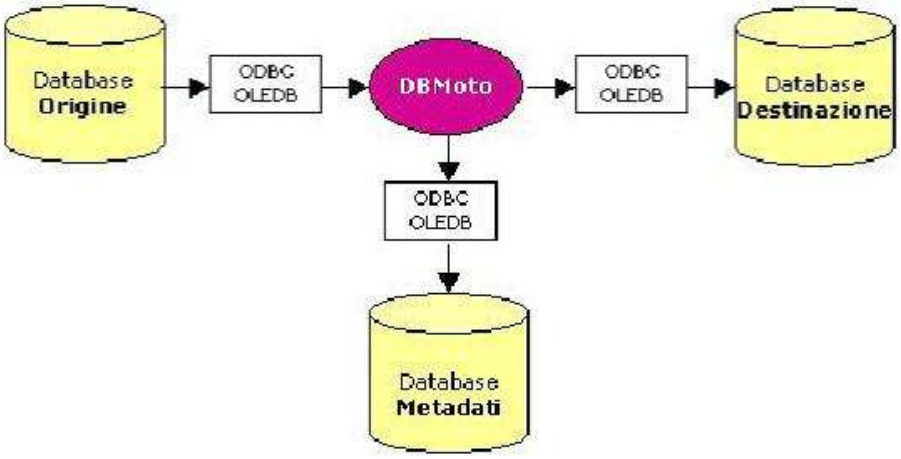

Figure 4.7: DBMoto OLEDB and ODBC connections

connecting DBMoto to the Metadata database, "Oracle Provider for OLE DB" for connecting DBMoto to the source database and "MyODBC" driver for the connection to the target database.
It is necessary to set the "don't optimize column width" option in the setting for MyODBC, for DBMoto to work correctly.

To configure our test environment we have done the following steps:

- we have create the Metadata database schema using the "Getting Started Wizard";
- we have add a source database with the "Source Connection Wizard", in which we select the tables "GRCSS" for the replication process too;
- we have selected the target database with the "Target Connection Wizard";
- with "Create Target Table Wizard" we have generateed the script that creates the target table "GRCSS" with the same structure and the same key that this table has in the source database.

It is necessary that the *my.cnf* file of the MySQL server contains the line "sql_mode=ANSI_QUOTES", so that the scripts, that DBMoto produces, work properly.

For starting the replication configuration, we have activated the target table menu, as showing in figure 4.8.

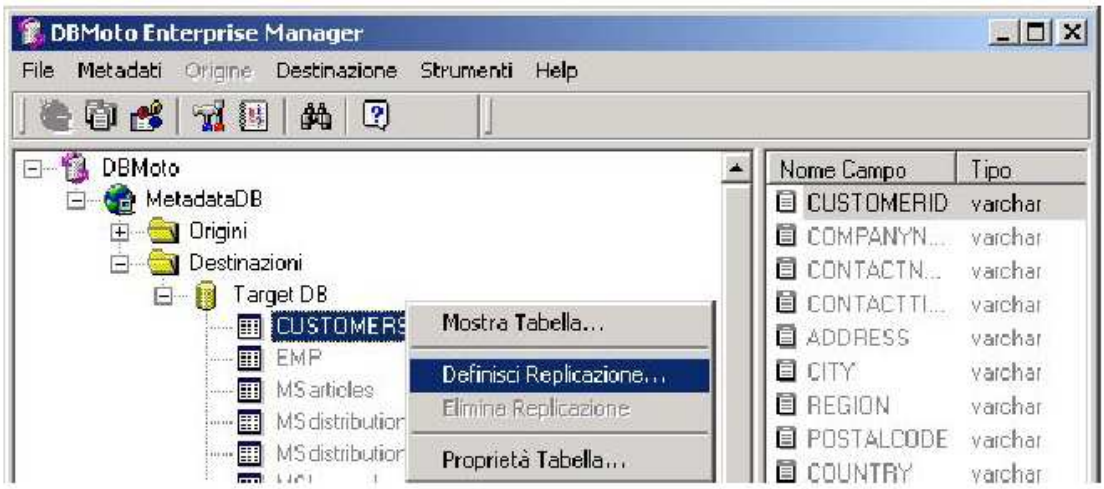

Figure 4.8: DBMoto replication setting

A user frame is presented on the screen (figure 4.9).

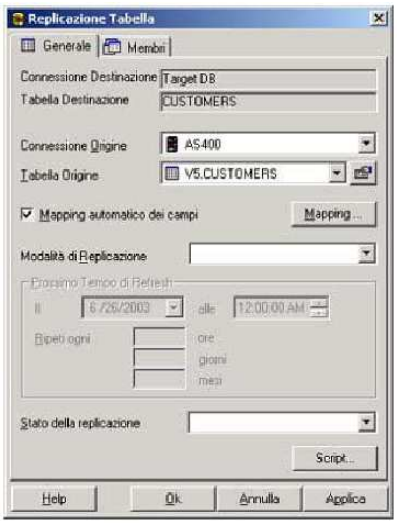

Figure 4.9: DBMoto replication setting

We have defined the parameters of the replication:

- `Target Connection`= MySQL;
- `Target Table` = GRCSS;
- `Source Connection` = Oracle;
- `Source Table` = GRCSS;

- setting `Automatic mapping`;
- `Replication Mode` = 2-Continuous mirroring;
- `Replication Status` = 0-Refresh is run;

In this way we have planned an initial *refresh* phase in which all the data in the source database are replicated in the target database, and a second phase of *mirroring* in which, on the base of the log file, the changes (`INSERT, UPDATE, DELETE`) happened in the time since the last reading on the source database, are carried on the target database (figure 4.10).

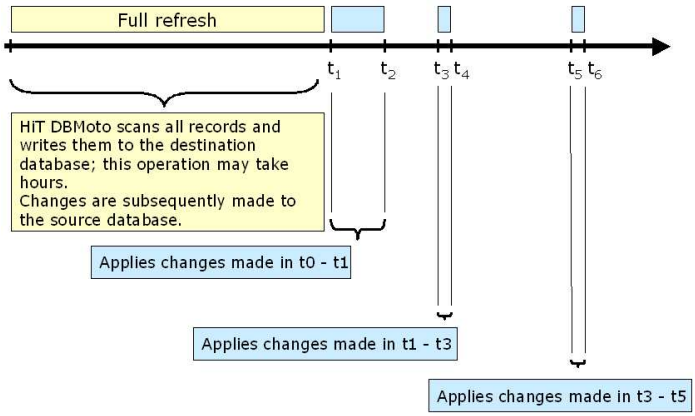

Figure 4.10: DBMoto mirroring

We had configured DBMoto for accessing the Oracle Redo log files (section 5.1) as shown in figure 4.11. The Oracle Redo logs are binary files, then DBMoto needs

Figure 4.11: DBMoto Oracle Redo Log setting

a dictionary (section 5.1) to interpret them. The user may specify the path of an existing dictionary or tell DBMoto to create it by querying the Oracle server.
For Oracle*10g* this dictionary is an online dictionary (figure 4.12). For activating

Figure 4.12: DBMoto Oracle Redo Log setting

this option it has been necessary to install a patch [22].

The Oracle Redo Log stores transactions executed on the database. Each record of the Redo Log is identified by a progressive number SCN (sequence change number).
Then we have configured DBMoto so that it reads from metadata the first SCN to process, it acquires the logs and saves the last SCN on the Metadata database (figure 4.13).

Once we have set the replication rules we have started the DBMoto Replicator as a Windows application; from the icon that appears in the toolbar we can start or stop the program (figure 4.14). DBMoto Replicator can run as a Windows service too.

We have tried to update the row of the source table with `INSERT, UPDATE` and `DELETE` operations and we have seen that the changes are propagated on the target

Figure 4.13: DBMoto Oracle Redo Log setting

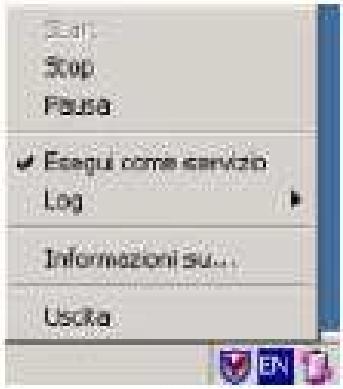

Figure 4.14: DBMoto Replicator

table with minimal delay; reading the MySQL log file we have seen that DBMoto indeed propagates only the changes.

4.2.2 Comments

DBMoto performs refresh and real-time data replication. All major database platforms are supported.

It has a user-friendly graphical interface; administrative wizards allow an easy configuration and setup.
Configuration and management of the entire replication process are performed from a Windows platform.

Reports about applications of DBMoto are available from:

- *Phisician Associates*[8], that deploys a web-based medical management software for physician referral replicating critical data via DBMoto. They replicate iSeries data to SQL server in real-time
- *ASL Modena*, that needed to improve patient data transfer among the various hospitals in their organization. They link nine remote SQL servers with a centralized AS/400 server via DBMoto.

[8] A network of over 800 independent physicians in private practice near Pasadena.

The necessity of running the MySQL server with the ANSI_QUOTES options set may interfere with other applications that use the same server.

When we use the wizard for creating the target table, DBMoto automatically proposes the conversion of the column types. In our test the `VARCHAR2` Oracle type has been translated in `TEXT` type for the MySQL table; in this way, when we have tried to visualize the target table with DBMoto we could not see the field for a common problem of ADO (Activex Data Object) in reading `TEXT` fields.

Chapter 5

Oracle log extraction tools

This chapter provides an overview about the Oracle *Redo Log* files and Oracle log extraction tools: **LogMiner** and **Oracle Streams**. We also describe **Oracle Heterogeneous Connectivity**, an Oracle utility for information integration in a heterogeneous environment.
Most of the material in this chapter, including figures, is taken from [23, 24, 25, 26, 27].

5.1 Oracle Redo Log

5.1.1 Introduction to Oracle Redo Log

All changes made to the Oracle database are stored in two or more preallocated files, called **Redo Log** [23], when they occur. The Redo Log, then, protects the database in case of instance[1] failure.

Each database instance has its own **Redo Log groups**, that are called an instance **thread** of Redo. In a typical configuration only one thread is present and only one database instance accesses the Oracle Database. But you can have multiple instances that access a single database and each istance has its own thread of Redo.

You can choose to run an Oracle database in:

- `NOARCHIVELOG` mode, you disable the archiving of the Redo Log; only the most recent changes made to the database are available for instance recovery.
- `ARCHIVELOG` mode, you enable the archiving of the redo log.

5.1.2 Redo Log contents

Redo Log files are filled with **redo records** (also called **redo entries**).
Redo records record data that you can use to reconstruct all changes made to the

[1] A set of memory and process structures, running on a specific computer that are responsible for translating the SQL calls given by the client, to effect data transfers to and from the database stored in the operating system files.

database.
They are buffered in a circular way in the Redo Log buffer and they are written to one of the Redo Log files by the **Log Writer** (LGWR)[2] whenever a transaction is committed. When this process writes them into the files, it assigns a **system change number** (SCN) to identify the redo records for each committed transaction.

When the current Redo Log file fills, LGWR writes to the next available Redo Log file. When the last available Redo Log file is filled, LGWR return to the first Redo Log file and writes to it, starting the cycle again.
The Redo Log on which LGWR is writing is called the **current** Redo Log file.
The point at which the database stops writing to one redo log and starts writing to another is called **log switch**. When a log switch occurs the Oracle database assigns a new **log sequence number** to each Redo Log file. In this way, when a crash occurs, the database applies Redo Log files in ascending order using the log sequence number.
A database requires, then, a minimum of two Redo Log files to guarantee that one is always available for writing while the other is being archived.
If Redo Log files are multiplexed, LGWR concurrently writes the same information to multiple identical Redo Log files.
If the database is in `ARCHIVELOG` mode, there is an archive process that takes a filled Redo Log file and copies it to a different device, either a separate disk or directly to a tape.

5.2 LogMiner

LogMiner [24] is a part of the Oracle database, that enables you to query online and archived Redo Log files through an SQL interface.

LogMiner can be used to track any data manipulation language (DML) and data definition language (DDL) statements executed on the database, the order in which they were executed and who executed them.

The **LogMiner dictionary** allows LogMiner to provide table and column names, instead of internal object IDs, when it shows the Redo Log data that you request.

Steps in a typical LogMiner session are the following:

1. Specify a LogMiner dictionary, using the `DBMS_LOGMNR_D.BUILD` procedure.
2. Specify a list of Redo Log files for analysis, using the `DBMS_LOGMNR.ADD_LOGFILE` procedure.
3. Start LogMiner, using the `DBMS_LOGMNR.START_LOGMNR` procedure.
4. Request the redo data of interest, querying the `V$LOGMNR_CONTENTS` view.
5. Stop the LogMiner, using the `DBMS_LOGMNR.END_LOGMNR` procedure.

[2] A database background process.

5.2.1 The V$LOGMNR_CONTENTS view

This is a view of the SYS schema that provides information about changes made to the database.

If the change was due to a DML statement, the reconstructed SQL statement shows (in the SQL_REDO column) the statement used to generate redo records, and shows the statement needed to undo the change (in the SQL_UNDO column).

LogMiners offers features for formatting the data that is returned to V$LOGMNR_CONTENTS. For example you can request only the rows belonging to committed transactions, or you can format reconstructed SQL statements for reexecution.

5.3 Oracle Streams

5.3.1 Introduction to Oracle Streams

Oracle Streams [25] enable information sharing. Using Oracle Streams, you can propagate information within a database or from one database to another.

Streams can capture, stage and manage events (as DML changes or DDL changes) in the database automatically. You can also put user-defined events into a stream. When events reach the destination, Streams can consume them.
The Streams information flow can be then represented in the following way:

$$CAPTURE \rightarrow STAGING \rightarrow CONSUMPTION$$

The Oracle Streams system consists of three main processes:

1. **Capture**, that captures changes to database objects from Redo Log. These changes are placed in a queue.

2. **Propagate**, that propagates changes from a queue at the source database to a queue at the destination database.

3. **Apply**, that retrieves changes from the destination queue area and applies them to a database.

When a capture process captures events, it is referred to as **implicit capture**; when user and application may enqueue event into a queue manually, it is referred to as **explicit capture**.

5.3.2 Capture Process

A **capture process** is an optional background process whose process name is *cnnn*, where *nnn* is a capture process number (from *c001* to *c999*).

This process scans the database Redo Log to capture DML and DDL changes made to the database objects. The database where the changes were generated is called the **source database**.

A capture process can be associated with only one user, but one user may be associated with many capture processes.

A capture process consists of the following components that are executed in parallel:

- **Reader server**, that reads the Redo Log and divides it into regions.
- **Preparer servers**, one or more, that scan the Redo Log's regions in parallel and perform prefiltering[3] of changes found.
- **Builder server**, that merges redo records from the preparer servers and passes them to the capture process.

When the capture process receives merged redo records:

1. it formats each captured changes into an event called a **Logical Change Record** (LCR);
2. eventually sends the LCRs to the rule engine for full evaluation;
3. enqueues the LCRs or discards them.

At a **checkpoint** the capture process records its current state persistently in the data dictionary of the database in which the process run.

5.3.2.1 Logical Change Records

There are two types of LCRs: **row LCRs** and **DDL LCRs**.

After capturing an LCR, a capture process enqueues an event containing the LCR into a `SYS.AnyData`[4] queue.

Row LCRs

A row LCR describes a change to the data in a single row or a change to a single `LONG`, `LONG RAW`, or LOB[5] column in a row; this change results from a data manipulation language (DML) statement or a piecewise update to a LOB.

Each row LCR is encapsulated in an object of `LCR$_ROW_RECORD` type and contains the following attributes:

- `source_database_name`, the name of the source database where the change occurred;
- `command_type`, the type of DML statement that produced the change;

[3] Sendig partial information about changes to the rules engine (section5.3.2.2) for the evalutation and receiving the result.

[4] These queue can stage events of different type.

[5] Large OBject datatype that let you store blocks of unstructured data (such as text, graphics images, video clips, and sound waveforms) up to four gigabytes in size.

- `object_owner`, the name of the schema that contains the table with the changed row;
- `object_name`, the name of the table with the changed row;
- `tag`, a row tag that can be used to track the LCR;
- `transaction_id`, the identifier of the transaction in which the DML statement was run;
- `scn`, the System Change Number (SCN) at the time when the change record was written in the Redo Log;
- `old_values`, the old column values related to the change;
- `new_values`, the new column values related to the change.

DDL LCRs

A DDL LCR describes a data definition language (DDL) change.

A DDL LCR contains the following attributes:

- `source_database_name`, the name of the source database where the change occurred;
- `command_type`, the type of DDL statement that produced the change;
- `object_owner`, the schema name of the user who owns the database object on which the DDL statement was run;
- `object_name`, the name of the database object on which the DDL statement was run;
- `object_type`, the type of the database object on which the DDL statement was run;
- `ddl_text`, the text of the DDL statement;
- `logon_user`, the user whose session executed the DDL statement;
- `current_schema`, the schema that is used if no schema is specified for an object in the DDL text;
- `base_table_owner`, owner of the table on which the DDL statement is dependent;
- `base_table_name`, name of the table on which the DDL statement is dependent;
- `tag`, a row tag that can be used to track the LCR;

- `transaction_id`, the identifier of the transaction in which the DDL statement was run;
- `scn`, the System Change Number (SCN) at the time when the change record was written in the Redo Log.

5.3.2.2 Capture Process Rule

A capture process either captures or dicards changes based on rules[6] that you define.

These rules can belong to a positive or negative rule set. The negative rule set is always evaluated[7] first.

You can specify capture process rules at the following level:

- table rule, captures or discards row changes to a particular table;
- schema rule, captures or discards row changes to the database object in a particular schema;
- global rule, captures or discards row changes in the database.

5.3.2.3 Local capture and Downstream capture

A capture process may run on the source database or on a remote database.

Local Capture

The capture process runs on the source database (figure 5.1). The first time a capture process is started, the extracted data dictionary information in the Redo Log is used to create a **LogMiner data dictionary**. Additional capture processes may use this LogMiner dictionary or they may create new LogMiner data dictionaries.

The capture process scans the Redo Log for changes using LogMiner, because the information in the primary data dictionary may not apply to the changes being captured from the Redo Log. These changes may have occurred days before they are captured by a capture process.

[6] A **rule** is a database object that enables a client to perform an action when an event occurs and a condition is satisfied.

[7] Rules are evaluated by a **rule engine**, which is a built-in part of Oracle.

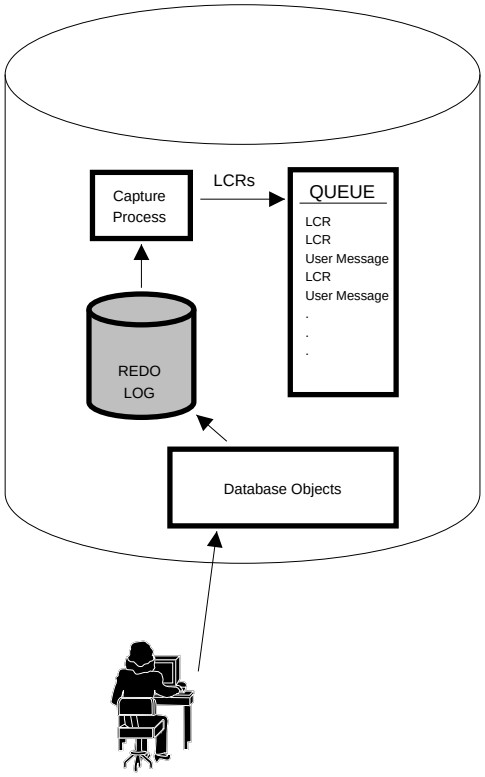

Figure 5.1: Local capture

Downstream Capture

The capture process runs on a remote database called **downstream database** (figure 5.2).

Archived Redo Log files from the source database are copied to the downstream database, and the capture process captures changes in these files at the downstream database.

The first time a downstream capture process is started, data dictionary information in the Redo Log is used to create a LogMiner data dictionary at the downstream database.

5.3.2.4 Rule-Based Transformations

A **rule-based transformation** is any user defined modification to an event that results when a rule in a positive rule set evaluates to `TRUE`.

You can modify both LCRs and user messages. If the LCR event is transformed during capture, the transformed LCR event is enqueued into the queue used by the capture process (figure 5.3).

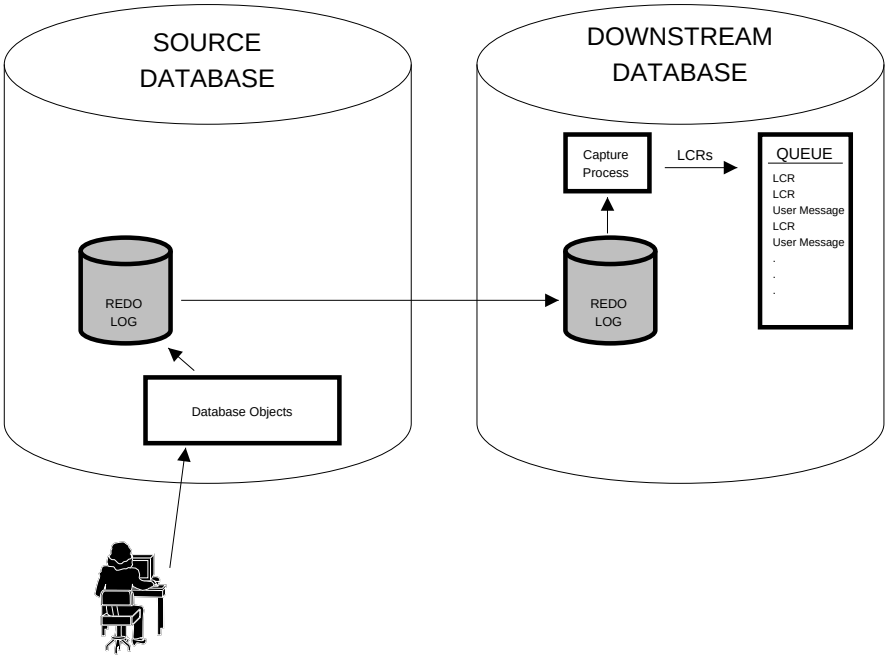

Figure 5.2: Downstream capture

5.3.2.5 Managing a Capture Process

Creating a Capture Process

You can use any of the following procedures to create a local capture process[8]:

- `DBMS_STREAMS_ADM.ADD_TABLE_RULES`
- `DBMS_STREAMS_ADM.ADD_SUBSET_RULES`
- `DBMS_STREAMS_ADM.ADD_SCHEMA_RULES`
- `DBMS_STREAMS_ADM.ADD_GLOBAL_RULES`
- `DBMS_CAPTURE_ADM.CREATE_CAPTURE`

The `CREATE_CAPTURE` procedure creates a capture process but does not create a rule set or rules for the process.

Starting a Capture Process

You can use the `DBMS_CAPTURE_ADM.START_CAPTURE` procedure to start an existing capture process.

[8]You can create a script and run it with SQL*PLUS (Oracle tool).

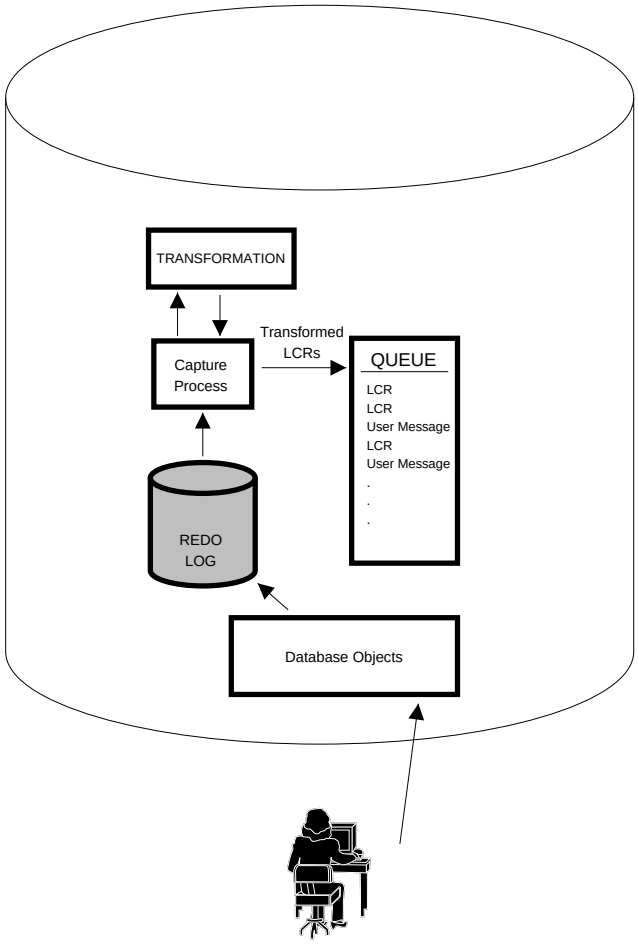

Figure 5.3: Transformation during capture

Stopping a Capture Process

You can use the `DBMS_CAPTURE_ADM.STOP_CAPTURE` procedure to stop an existing capture process.

5.3.3 Streams Staging and Propagation

In a `SYS.AnyData` queue there are two type of events:

- **captured event**, containing an LCR captured by a capture process;
- **user-enqueue event**, containing a user message (LCR or other type of messages) created by users or applications.

Staged events can be consumed or propagated, or both.

The queue to which events are propagated can reside in a different database than the queue from which the events are propagated, or in the same database. In

either case the first queue is called **source queue** and the second queue is called **destination queue**. There can be a one-to-many, many-to-one or many-to-many relationship between source and destintion queues.

Propagation process enqueues the event into the destination queue. An event may be propagated to a number of queues before it arrives at a destination database.

5.3.3.1 Event Propagation

Propagation is always between two queues, but a single queue may partecipate in many propagations.
A propagation may propagate all of the events in a source queue to a destination queue, or a propagation may propagate only a subset of events.

When an event is propagated with success between two queues, the destination queue acknowledges propagation of event. In the case of multiple destination queues, the event remains in the source queue until each destination queue acknowledges the propagations to the source queue.

5.3.3.2 Propagations Rules

A propagation either propagates or discards events based on rules that you define. These rules can belong to a positive or negative rule set. The negative rule set is always evaluated first.

You can specify propagation rules for LCR events at the following level:

- table rule, propagates or discards row changes to a particular table;
- schema rule, propagates or discards row changes to the database object in a particular schema;
- global rule, propagates or discards row changes in the source queue.

You can create rules to control propagations for non-LCR events.

5.3.3.3 Directed Networks

A *directed network* is one in which propagated events may pass through one or more intermediate databases before arriving at the destination database where they are consumed. An event may (**queue forwarding**) or may not (**apply forwarding**) be processed by an apply process at an intermediate database.

5.3.3.4 Propagations Jobs

A propagations job is the mechanism that propagates events from a source queue to a destination queue[9]. It may be used by more than one propagation.

[9] A job can be configured using the `DBMS_JOBS` package.

5.3.3.5 Stream Data Dictionary for Propagations

Propagation uses a **Stream Data Dictionary** to keep track of database objects from a particular source database.

When a database object is prepared for istantiation at a source database, a Stream data dictionary is populated automatically at the database where changes to the object are captured by a capture process.
The information in this dictionary at the source database is needed to evaluate rules at any database that propagates the captured events from the source database.

5.3.3.6 Rule-Based Transformations

A transformation can be performed during propagation (figure 5.4). It happens during dequeue.

Some destination queues can receive a transformed event, while other destination queues can receive the original event.

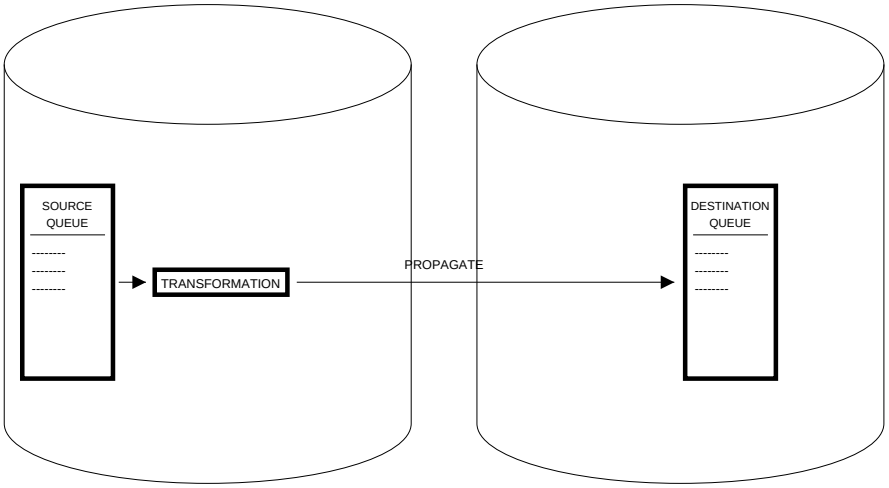

Figure 5.4: Transformation during propagation

5.3.3.7 Managing Streams Propagation

Creating a Propagation

Yuo can use any of the following procedures to create a propagation process:

- `DBMS_STREAMS_ADM.ADD_TABLE_PROPAGATION_RULES`
- `DBMS_STREAMS_ADM.ADD_SUBSET_PROPAGATION_RULES`
- `DBMS_STREAMS_ADM.ADD_SCHEMA_PROPAGATION_RULES`

- `DBMS_STREAMS_ADM.ADD_GLOBAL_PROPAGATION_RULES`
- `DBMS_CAPTURE_ADM.CREATE_PROPAGATION`

The `CREATE_PROPAGATION` procedure creates a propagation process but does not create a rule set or rules for the propagation.

Enabling a Propagation Job

You can use the `ENABLE_PROPAGATION_SCHEDULE` procedure to enable a propagation job.

Disabling a Propagation Job

You can use the `DISABLE_PROPAGATION_SCHEDULE` procedure to stop an existing propagation job.

5.3.4 Apply Process

An **apply process** is an optional Oracle background process that dequeues LCRs and user messages from a specific `SYS.AnyData` queue and either applies each one directly or passes it as parameter to a user-defined procedure.

An apply process consist of the following components:

- **Reader server**, dequeues events and assembles them into transactions, then returns the assembled transactions to the coordinator process.
- **Coordinator process**, sends the transactions to the apply server. This process is an Oracle background process whose process name is `a`*`nnn`*, where *nnn* is a coordinator process number (from `a`*`001`* to `a`*`999`*).
- **Apply server**, one or more, that apply the events or pass them to the appropriate handler; if an apply server encounters an error and does not resolve it, then it rolls back the transaction and places the entire transaction in the error queue.

5.3.4.1 Apply Process Rules

An apply process retrieves events from a queue or discards them based on rules. These rules can belong to a positive or negative rule set. The negative rule set is always evaluated first.

You can specify propagation rules for LCR events at the following level:

- table rule, applies or discards row changes to a particular table;
- schema rule, applies or discards row changes to the database object in a particular schema;

- global rule, applies or discards row changes in the queue associated with an apply process.

You can create rules to control apply process behavior for non-LCR events.

5.3.4.2 Event Processing with an Apply Process

A single apply process can apply either captured events or user-enqueued events, but not both.

LCR Event Processing

An apply process can apply captured LCRs from only one source database. If there is a single queue that contains captured LCRs from multiple source databases, then there must be multiple apply process retrieving these LCRs.

A single apply process can apply user-enqueued events, containing LCRs, even if they are from multiple source databases.

You can configure an apply process to process a captured LCR or an user-enqueued event that contains an LCR in the following way:

- **Direct apply**, apply the LCR event directly; the apply process either successfully applies the change in the LCR to a database object or, if a conflict on an apply error is encountered, tries to resolve the error with a conflict handler or a user-specified procedure called an error handler.
- **Custom apply**, call a user procedure to process the LCR event; the apply process passes the LCR event as a parameter to a user procedure.

  This procedure can be:

  - a **DML handler**, that processes row LCRs resulting from DML statements;
  - a **DDL handler**, that processes DDL LCRs resulting from DDL statement.

  An apply process can have many DML handlers but only one DDL handler.

Non-LCR User Message Processing

This event is processed by the **message handler** specified for an apply process.

5.3.4.3 Stream Data Dictionary for an Apply Process

The mapping information in the stream data dictionary at the source database is needed to interpret the contents of LCR at any database that apply the captured event. Oracle automatically propagates relevant information from this dictionary at the source database to all other databases that apply captured events from the source database.

5.3.4.4 Error Queue

This queue contains all of the current apply errors encountered at the local destination database only; it does not contain information about errors for apply processes running in other databases in a Streams enviroments.

5.3.4.5 Rule-Based Transformations

A transformation can be performed during apply (figure 5.5). It happens during dequeue.

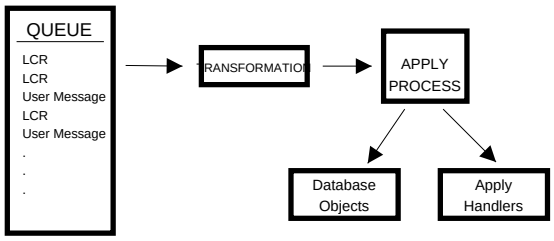

Figure 5.5: Transformation during apply

5.3.4.6 Managing an Apply Process

Creating an Apply Process

You can use any of the following procedures to create an apply process:

- `DBMS_STREAMS_ADM.ADD_TABLE_RULES`
- `DBMS_STREAMS_ADM.ADD_SUBSET_RULES`
- `DBMS_STREAMS_ADM.ADD_SCHEMA_RULES`
- `DBMS_STREAMS_ADM.ADD_GLOBAL_RULES`
- `DBMS_STREAMS_ADM.ADD_MESSAGE_RULES`
- `DBMS_APPLY_ADM.CREATE_APPLY`

The `CREATE_APPLY` procedure creates an apply process but does not create a rule set or rules for the process.

Starting an Apply Process
You can use the `DBMS_APPLY_ADM.START_APPLY` procedure to start an existing apply process.

Stopping an Apply Process
You can use the `DBMS_APPLY_ADM.STOP_APPLY` procedure to stop an existing apply process.

Setting an Apply User
You can use the `DBMS_APPLY_ADM.ALTER_APPLY` procedure to set the apply user for an apply process.

Set an Apply Process Parameter
You can use the `DBMS_APPLY_ADM.SET_PARAMETER` procedure to set an apply process parameter. These parameters control the way an apply process operates.

5.3.5 Replication using Oracle Streams

Oracle Streams enables **replication** [26]. Replication is the process of sharing database objects and data at multiple databases. A change to one of these objects is shared with the other databases.
You use rules to control the information flow in a Streams replication enviroment.

5.3.5.1 Conflict resolution

Conflicts are possibile in a Streams enviroments where data is shared between multiple databases. In this case you can have concurrent DML changes on the same data at multiple databases.
Streams automatically detects conflicts and tries to resolve them.

Conflicts Types
There are the following type of conflict:

- **Update conflict**, occurs when the apply process applies a row LCR containing an update to a row that conflicts with another update to the same row. An apply process detects this type of conflict if there is any difference between the old values for a row in a row LCR and the current values of the same row at the destination database.

- **Uniqueness conflict**, occurs when the apply process applies a row LCR containing a change to a row that violates a uniqueness integrity constraint. An apply process detects this type of conflict if a uniqueness constraint violation occurs when applying an LCR that contains an insert on an update operation.

- **Delete conflict**, occurs when two transaction originated at differrent databases, with one transaction deletes a row and another transaction updates or deletes the same row. An apply process detects this type of conflict if it can not find a row when applying an LCR that contains an insert or update operation, because a primary key of the row does not exist.

- **Foreign key conflict**, occurs when the apply process applies a row LCR containing a change to a row that violates a foreign key constraint. An apply process detects this type of conflict if a foreing key constraint violation occurs when applying an LCR.

Streams provides prebuilt conflict handlers to resolve update conflicts but not the other type of conflicts. A conflict handler is applied as soon as a conflict is detected.

5.3.5.2 Streams Tags

There is a **tag** associated with every redo entry. A tag can be used to determine whether an LCR contains a change that originated in the local database or at a different database, or for other LCR tracking purposes.

In a stream replication enviroment you can use tags to avoid change cycling, that is a change that come back to the database where it originated.

There are various Streams enviroments that produced change cycling:

-Each database is a source and destination database for every other database. Each database communicates directly with every other database (figure 5.6). In this case you can:

- configure one apply process[10] at each database to generate non-`NULL` Redo Log tags for change from each source database.

- configure the capture process at each database to capture changes only if the tag in the Redo Log entry for the change is `NULL`.

In this configuration all the changes applied by the apply processes are never recaptured. All databases are synchronized.

-Primary database shares data with several secondary databases. The secondary databases share data only with the primary database (figure 5.7). The secondary database has one apply process that applies changes from the primary database. In this case you can :

- at the primary database, configure each apply process to generate non-`NULL` redo tags that indicate the site from which it is receiving changes, and configure the capture process to capture changes regardless of the tag.

[10] An apply process generates entries in the Redo Log of a destination database when it applies DML or DDL changes.

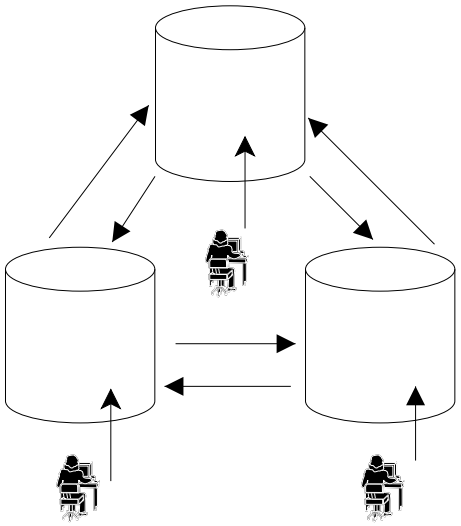

Figure 5.6: Each database is a source and a destination database

- at each secondary database, configure each apply process to generate non-`NULL` redo tags and configure the capture process to capture changes only if the tag in the Redo Log entry is `NULL`.
- configure a propagation from the primary database to each secondary databases; each propagation propagate all LCRs except for changes that originated at the secondary database.
- configure one propagation from the each secondary databases to primary database.

No changes are lost in this enviroment and all databases are synchronized.

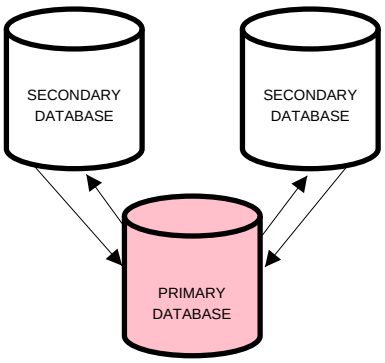

Figure 5.7: Primary database shares data with several secondary databases

-Primary database shares data with several extended secondary databases. In this case the secondary databases have other secondary databases, called **remote secondary databases**, connected to them (figure 5.8). A remote secondary database can share data with the primary database through a secondary database. In this enviroment you can :

- at the primary database, configure each apply process to generate non-`NULL` redo tags that indicate the site from which it is receiving changes, and configure the capture process to capture changes regardless of the tag.
- at each remote secondary database, configure each apply process to generate non-`NULL` redo tags and configure the capture process to capture changes only if the tag in the Redo Log entry is `NULL`.
- at each remote secondary database, configure one apply process to apply changes from the primary database with a redo tag value that is equivalent to the hexadecimal value '00'[11]. Furthermore configure one apply process to apply changes from each of its remote secondary databases and configure a capture process that capture all changes to the shared data in the Redo Log, regardless of the tag value.
- configure propagation from each secondary database to the primary database, that propagate all LCRs except for changes originated at the primary database.
- configure propagation from each secondary database to each remote secondary database, that propagate all LCRs except for changes originated at the remote secondary database.

[11] This is the default tag value for an apply process.

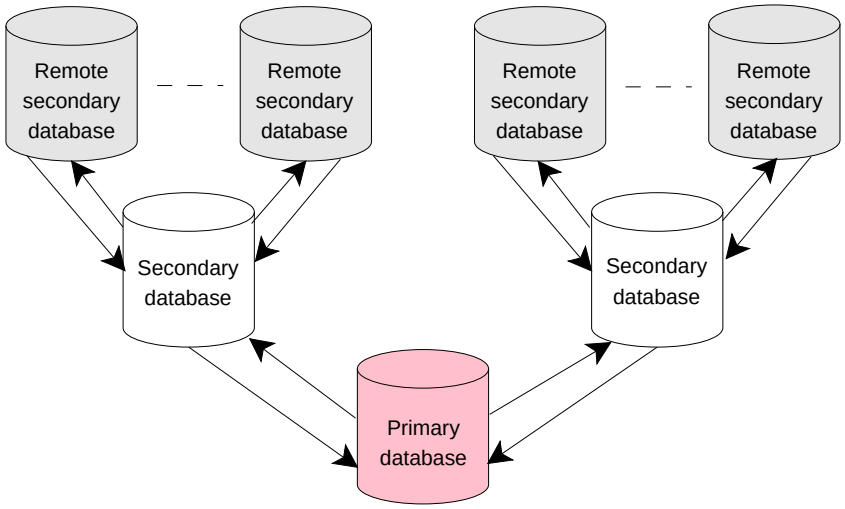

Figure 5.8: Primary database shares data with several extended secondary databases

5.3.6 Heterogeneous data sharing with Streams

5.3.6.1 Oracle to Non-Oracle

With Streams you can have an Oracle to Non-Oracle data sharing (figure 5.9).

The events intended for the non-Oracle destination database are dequeued in the Oracle database itself and an apply process at the Oracle database applies the changes to the non-Oracle database across a network connection through an Oracle gateway (section 5.4).

In this type of environment the capture process functions the same way as it would in a Oracle-only environment. Parallel apply to non-Oracle database are not supported.

An apply process can apply only the following types of DML changes:

- `INSERT`
- `UPDATE`
- `DELETE`

The DDL change cannot be applied at Non-Oracle databases.

5.3.6.2 Non-Oracle to Oracle

In this case a user application is required (figure 5.10). The application must assemble and order the transactions of the non-Oracle database, and must convert them into LCR. Next must enqueue the LCRs into a queue in a Oracle database. The application is therefore responsible for change capture.

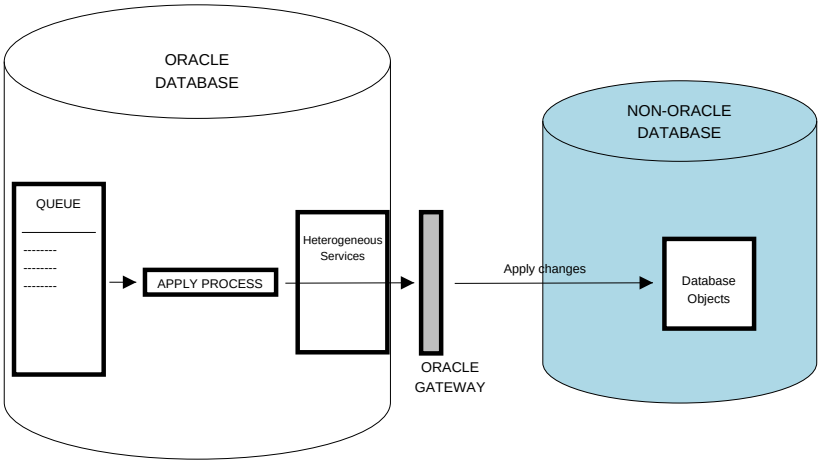

Figure 5.9: Oracle database shares data with a non-Oracle database

5.3.6.3 Non-Oracle to Non-Oracle

This environment is supported through a combination of non-Oracle to Oracle data sharing and Oracle to non-Oracle data sharing.

5.4 Heterogeneous platform support

5.4.1 Heterogeneous Connectivity

The two methods that Oracle uses for solving the heterogeneous connectivity [27] problem are:

- **Oracle Transparent Gateways**, each of these gateways has been designed for a particular non-Oracle system[12]. These gateways are separate from the Oracle server. They can be located on the non-Oracle system, the Oracle system or a stand-alone system.

- **Generic Connectivity**, set of agents that require drivers to provide access to the non-Oracle system. The functionality of Generic Connectivity is more limited that of Oracle Transparent Gateways, because connection can be made only to the local Oracle database server. Generic connectivity cannot perteci-pate in distributed transactions, it supports single-site transactions only.

The heterogeneous connectivity process is divided into two parts (figure 5.11):

- **Heteregeneous service**, that is integrated in the Oracle server.

[12]Sybase, MS SQL, Informix, Ingres, Teradata, RDB, RMS.

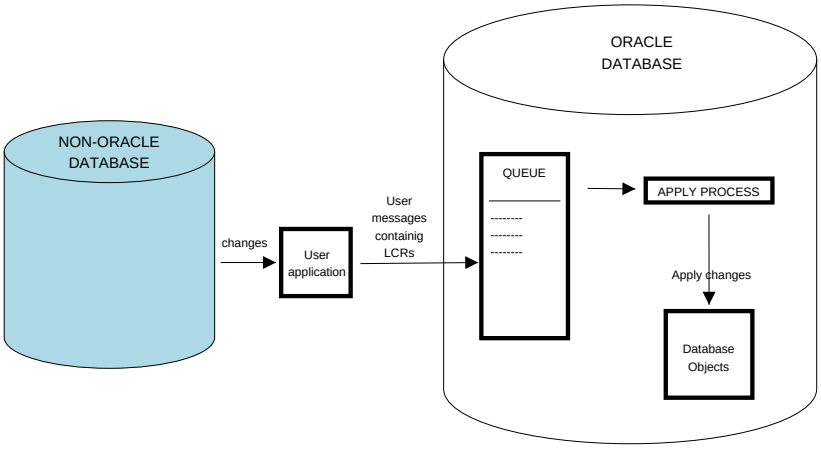

Figure 5.10: Non-Oracle to Oracle heterogeneous data sharing

- **Agent**, that provides connectivity to non-Oracle systems. The agent process consists of two components: agent generic code and the driver (the module that communicates with the non-Oracle system). Oracle Transparent Gateway and Generic Connectivity are two types of agents.

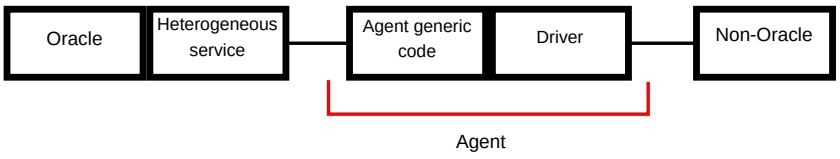

Figure 5.11: Heterogeneous Connectivity Architecture

Heteregeneous service provides the following service:

- **Transaction service**, that ensures an authenticated session between Oracle and non-Oracle system.
- **SQL service**, that provides SQL dialect translation and data dictionary translation.
- **Procedural service**, provides an interface for executing stored procedures on a non-Oracle system.
- **Pass-Through SQL**, that provides the possibility to execute functions and procedures on the non-Oracle system. It can be used to perform DDL change on the non-Oracle system.

Datatype translation is performed by the agent.

5.4.2 Generic Connectivity

Generic connectivity is implemented as either a Heterogeneous Services ODBC agent (that uses an ODBC driver to interface with the non-Oracle system) or a Heterogeneous Services OLE DB agent (whitch uses an OLE DB driver).

These agents have to be on the same machine as the Oracle database, they are part of the Oracle server. The ODBC or OLE DB drivers must be on the same platform as the agent.

Oracle and non-Oracle system can be on the same or on separate machine.

The Oracle database server converts the datatypes used in ODBC and OLE DB in Oracle datatypes. This type of agent supports the following statements:

- `INSERT`
- `DELETE`
- `UPDATE`
- `SELECT`

This solution is free and avaible with the Oracle database.

Chapter 6

Oracle to Mysql data sharing with Streams

In some Oracle documentation [28, 29] we have found that Streams can apply changes to a non-Oracle system via Transparent Gateway or Generic Connectivity. Since a Transparent Gateway for Mysql does not exists, we have tested Oracle Streams and Oracle Generic Connectivity compatibility. In this chapter we describe the results of our evaluation.

6.1 Setup Generic Connectivity Agent to Mysql

First we have configured and tested an Oracle Generic Connectivity to connect from an Oracle 10.1.0.2.0 database on Scientific Linux platform to MySQL 4.1.9 database running on a remote Red Hat Linux machine.

We had two machines with these configurations:

1.
   - Scientific Linux
   - Oracle 10.1.0.2.0
   - host name = `oracle-db-3.cr.cnaf.infn.it`
   - database name = `pisatest`

2.
   - Red Hat Linux
   - MySQL 4.1.9
   - host name = `pcgrid03.pi.infn.it`
   - database name = `RCC1`

On the RCC1 database we have set up the grant tables to allow the `laura` Oracle user to connect from machine `oracle-db-3.cr.cnaf.infn.it`.

6.1.1 Setting up ODBC Driver

To use an ODBC agent, you must have an ODBC driver manager installed on the same machine as the Oracle Server.
We have downloaded [30] and installed on `oracle-db-3.cr.cnaf.infn.it` the *unixODBC-2.11.11* driver manager.

Then, we have downloaded [31] and installed on `oracle-db-3.cr.cnaf.infn.it` the *MySQLODBC-2.50.39* driver.

We have modified the file `odbc.ini` for creating an entry for our MySQL server.

On `oracle-db-3.cr.cnaf.infn.it`, in our startup file we have set some environment variables with the path to the `odbc.ini` file and the path for ODBC driver library.

Finally, we have tested MySQL ODBC driver with a test tool called `isql`: from the `oracle-db-3.cr.cnaf.infn.it` machine we were able to connect with the `RCC1` database, to create and manage tables in this database.

6.1.2 Setting up Oracle Heterogeneous Services

On Oracle server we have edited the `listener.ora` (Oracle Listener control file) file, adding an alias for the Heterogeneous Agent. Then we have stopped and restarted the Oracle TNS listener.
An entry for the alias has been added a the `tnsnames.ora` file too.
We have also created an inizialitation file for the Generic Connectivity agent.

After the Heterogeneous Service configuration has been completed, we have tested it. We have first created a public database link[1], called `RCC1.PI.INFN.IT`, for connecting us the MySQL database.

From the SQL*Plus prompt we were be able to send with success `INSERT`, `DELETE`, `UPDATE`, `SELECT` commands for the tables in the RCC1 database[2].

6.2 Setup an Oracle Streams environment

To test if Oracle Streams can apply changes to a MySQL database through a Generic Connectivity, we have setup a simple Streams environmement.

The Oracle database must be in `ARCHIVELOG` mode.

We have created a simple table `GRCSS`, with the same columns, on the Oracle database (under the `SYS` schema) and on the MySQL database. We have tried to maintain the two tables synchronized, replicating with Streams any change that occurs on the Oracle table, in the MySQL table.

We have set some parameters (e.g. `global_names=true`) in the `init.ora` file.
We have created a new user (`STRMADMIN2`) with the appropriate privileges (e.g. to manage queues, create rules, etc...) as a Streams administrator.

[1] A connection between two physical database servers.

[2] In the sql statements we have used `<tablename>@RCC1.PI.INF.IT` to point at our MySQL tables.

We have also created a new tablespace to hold the logMiner tables on the Oracle database.

Since the apply process requires additional information for some actions we have configured supplemental logging for the `GRCSS` table.

To set up the Streams replication environment we have used the scripts that you can read in appendix B.

We have created the queue `STREAMS_QUEUE` in which the `STRMADMIN_CAPTURE` capture process enqueues LCRs and the `STRMADMIN_APPLY` apply process dequeues LCRs. Since we had only one queue in our configuration, the propagation process is not necessary.

We have configured the dblink apply process parameter to `RCC.PI.INFN.IT` so that the apply process applies changes to the MySQL database.
We have set the instantiation SCN[3] at the source table.
We have started the apply process and then the capture process.

With the Streams processes activated, we have tried to make some DML changes on the Oracle table.
We have seen that the capture process captured the changes correctly.
Instead the apply process was aborted with the following error message in output:

```
ERROR: Apply 'STRMADMIN_APPLY' has aborted with message
ORA-12801: error signaled in parallel query server
P000 ORA-28584: heterogeneous apply internal error
ORA-28584: heterogeneous apply internal error
ORA-02063: preceding line from RCC1.PI.INFN.IT
```

6.3 Comments

We have asked the Oracle Support about this error . The response from Oracle said that Streams needs to have an Oracle Transparent Gateway for the Oracle to Non-Oracle Heterogeneous Data Sharing and that Transparent Gateways are available only for SQLServer, SYBASE, DB2 and IBM DRDA.

[3]The Streams process capture and apply only changes happened after this SCN.

Chapter 7

Log extraction test with LogMiner

In this chapter we describe some tests about the Oracle Redo Logs analysis with LogMiner. We also describe a plan of an Oracle Logs extraction method to integrate in *Constanza*.

7.1 Example using LogMiner

If we want to keep track of all changes done on a database, we can periodically check if changes have been made and in this case we can record them in a file.

To do this, we must activate periodically a LogMiner session that verifies if changes have been made on data of our interest since the last check. If this is true we extract the changes and store them in a update file.

We have used LogMiner to extract from the Redo Log files the operations made to the `laura.grcss` table, as a normal user `laura`.

To use the `DBMS_LOGMNR` package, user `laura` has been granted the `EXECUTE_CATALOG_ROLE`. Moreover the user must be granted `SELECT` privileges and `SELECT ANY TRANSACTION` on the `V$LOGMNR_CONTENTS` view.

Since this work aims to be integrated in the *Constanza* **Log Watcher** component (chapter 3), that has been implemented in C++, we have tried to use the `DBMS_LOGMNR` package through the **Oracle C++ Call Interface** (OCCI) [32], an Application Programming Interface (API) that provides C++ applications access to data in an Oracle database. To deploy OCCI based application and to simplify OCCI installation, you need an **Oracle Instant Client**.

In the *main.cc* file, that you can read in appendix C, we have followed these basic steps:

- we have created an `Environment` class, in which all OCCI processing takes place;
- we have opened a stateless connection pool;

- we have created a statement object to execute SQL commands;
- we have executed SQL statements that:
  - specify a list of Redo Log files for analysis, using the `DBMS_LOGMNR.ADD_LOGFILE` procedure;
  - start LogMiner, using the `DBMS_LOGMNR.START_LOGMNR` procedure;
- we have performed an infinite loop in which:
  - we have checked if new changes have been made on the table `laura.grcss`;
  - when the previous condition is verified, we have extracted the changes and put them in a file;

We have tried to run the program and while it runs we have performed several changes on the `laura.grcss` table. The program has extracted correctly from the Redo Log file the operations performed on the table.

If we want to replicate the changes on another Oracle database, we can apply directly the update file.
If we want to replicate the changes on a database of another vendor (e.g. MySQL), we must translate the update file in the appropriate format. For example, if we performed an `INSERT`, `UPDATE` or a `DELETE` operation on the `laura.grcss` table, the statements extracted from the redo log are the following:

```
insert into "LAURA"."GRCSS"("GRCSID","ADDRESS","NAME","STATUS")
         values ('4','4','4','4');

update "LAURA"."GRCSS" set "STATUS" = '5'  where "GRCSID" = '4'
           and "ADDRESS" = '4' and "NAME" = '4' and "STATUS" = '4';

delete from "LAURA"."GRCSS" where "GRCSID" = '4' and "ADDRESS" = '4'
         and "NAME" = '4' and "STATUS" = '5';
```

In fact the syntax for referring a schema object [33] in an Oracle database is

```
[ schema. ] object [ .part ] [ @ dblink ]
```

where:

- **schema** is the schema containing the object. The schema qualifier lets you refer to an object in a schema other than your own. You must be granted privileges to refer to objects in other schemas.
- **object** is the name of the object.
- **part** is a part of the object, such as a column or a partition of a table. Not all types of objects have parts.

- **dblink**, lets you refer to an object in a database other than your local database.

For executing the same SQL statement on the MySQL server we must eliminate the **schema** name and the quotation marks. This can be done, for example, with the use of a simple perl script (see appendix C).

7.2 Redo Log extraction plan in *Constanza*

After this test we have prepared a plan to allow heterogeneous replication of databases in *Constanza*.

First we describe how we think that the Oracle Redo Log extraction could be integrated in the *Constanza* code.

Currently in the `static void* run (void* arg)` function (see appendix C), in the *Constanza* **LogFWatcher** class, the MySQL log file last modification time is checked periodically. When the file has been modified, **Log Watcher** extracts the changes and creates an update file using the `void Log_generator()` function (see appendix C).

For the Oracle Logs extraction, the **Log Watcher** should work in the same way. It should constantly query the `V$LOGMNR_CONTENTS` view to check if new changes have been made[1] on the tables that must be replicated and when a change is detected, it queries the same view to extract it.

We can consider the old class **LogFWatcher** as a pure virtual class to use as a base for specializations for specific vendors. Two specializations of the *Constanza* **LogFWatcher** class should be created: **ORAWatcher** and **MySQLWatcher**. The class diagram is shown in figure 7.1.

[1] Checking the Timestamp of the last statement executed on objects that must be replicated.

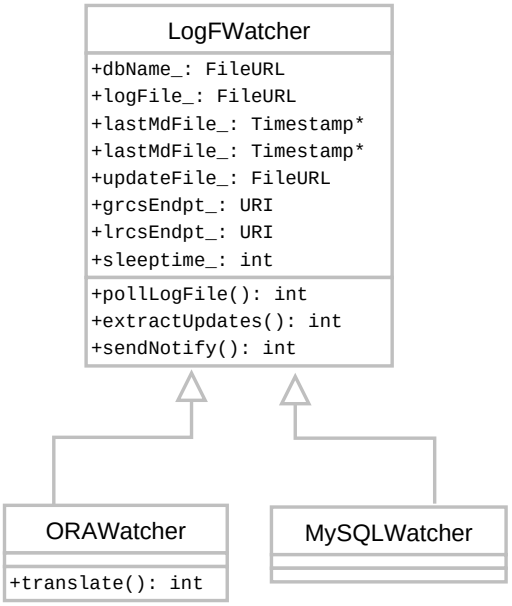

Figure 7.1: Class diagram

In the **ORAWatcher** subclass, the `int pollLogFile()` funtion checks if the log file has been modified. In figure 7.2 the flowchart of this function in the **ORAWatcher** class is shown. If an update has happened, the functiont returns 1, otherwise it returns 0.

When the function returns 1, the `int extractUpdates()` function is then called. This function executes a query to the `V$LOGMNR_CONTENTS` view to catch updates from the Oracle Redo Log. It saves the timestamp of the last transaction that has modified the database. The flowchart of this function is shown in figure 7.3.

Moreover a function for the update files translation, `int translate()`, must be developed, to guarantee that the SQL statements can be applied correctly to the replica database.

In the **MySQLWatcher** subclass, the `int pollLogFile()` funtion and the `int extractUpdates()` function make the same actions that currently are made in the *Constanza* **LogFWatcher** functions.

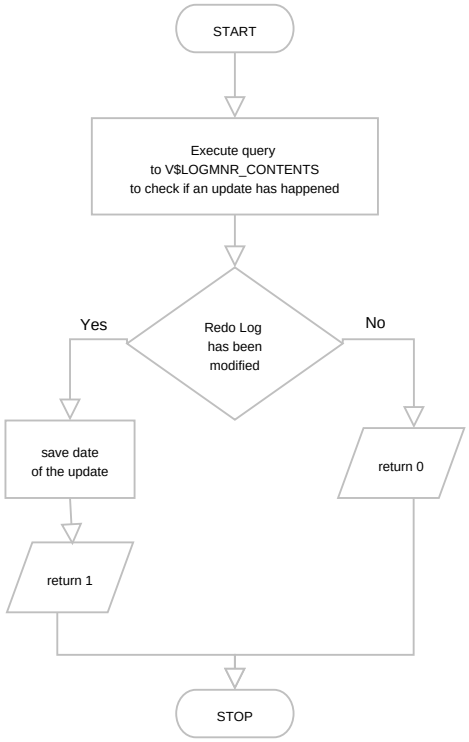

Figure 7.2: Flowchart of the `int pollLogFile()` function

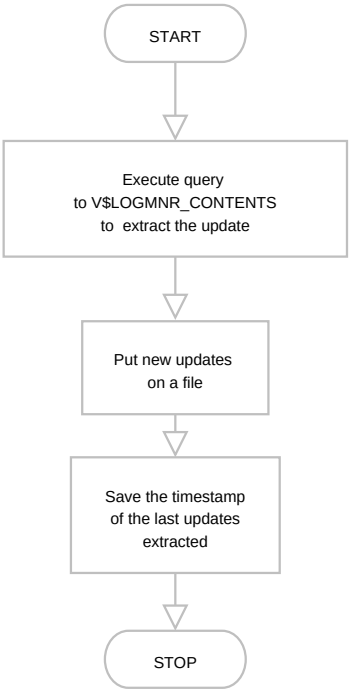

Figure 7.3: Flowchart of the `int extractUpdates()` funtion

Chapter 8

Conclusions

Currently, the requirement to maintain consistency in a Data Grid can be addressed in a Replica Consistency Service, such as *Constanza* (see chapter 3). The available solutions for the consistency problem in a Data Grid, however, do not take into account heterogeneity of data, that may be stored in different formats and in storage devices of different vendors.

In this thesis we have covered heterogeneous databases replication issues in a Grid environment. The focus has been on Oracle to Mysql databases replication, with the purpose of extending *Constanza* to a heterogeneous replication. In particular, we have investigated how to provide a mechanism to extract changes from database logs and apply them to replicas.

This problem requires tools for data integration. Then, we have examined some utilities provide by the **Oracle** server and two off-the-shelf tools, **Enhydra Octopus** and **DBMoto**.

The tests have shown that an optimum solution, already developed for a Grid environment, does not exist.

Enhydra Octopus currently provides Oracle to MySQL database replication but not propagation of incremental updates, since it performs replication by replacing the database files entirely. This is not suitable to replicate data in quantities in the order of Petabytes, as it happens in the Grids.

DBMoto has some limitations concerning the system requirements (e.g. it runs only on a Windows Server), the cost and the license.

Oracle logs extraction tests have involved **Oracle Streams** and **Oracle LogMiner**.

We have seen that Streams needs to have an Oracle Transparent Gateway for the Oracle to Non-Oracle Heterogeneous Data Sharing and that a Transparent Gateway is not available for MySQL.

LogMiner seems to be the best solution to integrate in *Constanza*. We have developed a plan that, through the **Oracle C++ Call Interface**, allows to use LogMiner in the *Constanza* code to extract update information from the Oracle log files. A module for the update translation in the appropriate format for Mysql must

be developed too.

Appendix A

Enhydra Octopus output files

A.1 LoaderJob.olj

```
<?xml version="1.0" encoding="UTF-8"?>
<loaderJob logMode="full" objectIDColumnName="next" objectIDIncrement="1"
objectIDTableName="objectid">
<jdbcDefaultParameters>
<jdbcSourceParameters dbVendor="Oracle" driverName="oracle">
<jdbcSourceParameter name="Password" value="***"/>
<jdbcSourceParameter name="Connection.Url"
   value="jdbc:oracle:thin:@oracle-db-3.cr.cnaf.infn.it:1521:pisatest"/>
<jdbcSourceParameter name="JdbcDriver" value="oracle.jdbc.driver.OracleDriver"/>
<jdbcSourceParameter name="User" value="***"/>
</jdbcSourceParameters>
<jdbcTargetParameters dbVendor="MySQL" driverName="mm">
<jdbcTargetParameter name="Password" value="***"/>
<jdbcTargetParameter name="Connection.Url" value="jdbc:mysql://localhost:3306/RCC1"/>
<jdbcTargetParameter name="JdbcDriver" value="org.gjt.mm.mysql.Driver"/>
<jdbcTargetParameter name="User" value="***"/>
</jdbcTargetParameters>
</jdbcDefaultParameters>
<definitionInclude>
<include href="xml/ImportDefinition.oli" parse="xml"/>
</definitionInclude>
</loaderJob>
```

A.2 ImportDefinition.oli

```
<definitionInclude>
<importDefinition name="GRCSS" tableName="GRCSS">
<valueColumns>
```

```
<valueColumn sourceColumnName="GRCSID" targetColumnName="GRCSID"
  targetTableID="0" targetTableName="GRCSS" valueMode="Key"/>
<valueColumn sourceColumnName="NAME" targetColumnName="NAME"
  targetTableID="0"  targetTableName="GRCSS" valueMode="Overwrite"/>
<valueColumn sourceColumnName="ADDRESS" targetColumnName="ADDRESS"
  targetTableID="0" targetTableName="GRCSS" valueMode="Overwrite"/>
<valueColumn sourceColumnName="STATUS" targetColumnName="STATUS"
  targetTableID="0" targetTableName="GRCSS" valueMode="Overwrite"/>
</valueColumns>
<tables>
<table insert="true" oidLogic="false" tableID="0" tableMode="Query"
  tableName="GRCSS"/>
</tables>
</importDefinition>
</definitionInclude>
```

A.3 Octopus Loader output

```
Default (Standard) Logger is load.

Application is started.
Loader loads XML file:
/home/users/iannone/octopus/usr/local/octopus-3.0/generatorOutput/Oracle/LoaderJob.olj

XML document is valid.

Import job No. 1 is started.
Importing and parsing XML file is started.
parseAndImport method is started.
importXMLFile method is started.
importXMLFile method is started.
importXMLFile method is finished.
parseAndImport method is finished.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
readConfigValues method is started.
readConfigValues method is finished.
readConfigValues method is started.
readConfigValues method is finished.
Importing and parsing XML file is finished.
loadSource method is started.
Import definition GRCSS is started at Mar 25, 2005 3:30:48 PM.
constantColumnTypes method is started.
```

```
constantColumnTypes method is finished.
targetColumnTypes method is started.
Query 'select GRCSID,NAME,ADDRESS,STATUS from  GRCSS'
                     will be executed
targetColumnTypes method is finished.
checkOidLogic method is started.
checkOidLogic method is finished.
querySource method is started.
querySource method is finished.
Query 'select GRCSID, NAME, ADDRESS, STATUS from GRCSS'
                     will be executed
Import block is started.
Working...  1. block of 1
Started from the 1. row.
insertTargetTable method is started.
Query 'select  GRCSID, NAME, ADDRESS, STATUS from GRCSS
                              where GRCSID = 4 ' will be executed
Query 'update GRCSS set NAME='prova4', ADDRESS='prova4', STATUS='up'
                              where GRCSID = 4 ' will be executed
insertTargetTable method is finished.
insertTargetTable method is started.
Query 'select  GRCSID, NAME, ADDRESS, STATUS from GRCSS
                              where GRCSID = 2 ' will be executed
Query 'update GRCSS set NAME='prova2', ADDRESS='prova2', STATUS='down'
                              where  GRCSID = 2 ' will be executed
insertTargetTable method is finished.
insertTargetTable method is started.
Query 'select  GRCSID, NAME, ADDRESS, STATUS from GRCSS
                              where GRCSID = 3 ' will be executed
Query 'update GRCSS set NAME='prova3', ADDRESS='prova3', STATUS='down'
                              where  GRCSID = 3 ' will be executed
insertTargetTable method is finished.
Import definition GRCSS is finished at Mar 25, 2005 3:30:50 PM.
 Duration of importDefinition:2,680 seconds
loadSource method is finished.
All rows are commited.
Application is finished.
All jobs duration: 5,456 seconds
```

A.4 Octopus Loader output

```
Default (Standard) Logger is load.
```

```
Application is started.
Loader loads XML file:
/home/users/iannone/octopus/usr/local/octopus-3.0/generatorOutput/Oracle/LoaderJob.olj

XML document is valid.
importVariable  method is started.
importVariable  method is finished.
importLoaderJobAttributes method is started.
importLoaderJobAttributes method is finished.
parseImportJob method is started.
parseImportJob method is finished.
parseCopyTable method is started.
parseImportJob method is finished.
parseSql method is started.
parseSql method is finished.
parseEcho method is started.
parseSql method is finished.
importRestartCounter method is started.
importRestartCounter method is finished.
parseMainElements method is started.
parseMainElements method is finished.
parseImportJDBCDefaultParameters method is started.
parseImportJDBCDefaultParameters method is finished.
readConfigValues method is started.
readConfigValues method is finished.

Import job No. 1 is started.
readConfigValues method is started.
readConfigValues method is finished.
Importing and parsing XML file is started.
importSQLStatement  method is started.
importSQLStatement  method is finished.
Importing and parsing XML file is started.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
Importing and parsing XML file is finished.
method executeSQLStatement is started
SQL statement DropTables is started
SQL_stmt : '

DROP TABLE GRCSS ;
```

```
        ' will be executed
SQL statement DropTables is finished
method executeSQLStatement is finished
 Duration :0,439 seconds

Import job No. 2 is started.
readConfigValues method is started.
readConfigValues method is finished.
Importing and parsing XML file is started.
importSQLStatement  method is started.
importSQLStatement  method is finished.
Importing and parsing XML file is started.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
Importing and parsing XML file is finished.
method executeSQLStatement is started
SQL statement CreateTables is started
SQL_stmt : '
            Create table GRCSS
(
GRCSID  NUMERIC (6,0) NOT NULL    ,
NAME  VARCHAR(20)     ,
ADDRESS  VARCHAR(40) NOT NULL    ,
STATUS  VARCHAR(10) NOT NULL
);


        ' will be executed
SQL statement CreateTables is finished
method executeSQLStatement is finished
 Duration :0,371 seconds

Import job No. 3 is started.
Importing and parsing XML file is started.
parseAndImport method is started.
importXMLFile method is started.
importXMLFile method is started.
importXMLFile method is finished.
parseAndImport method is finished.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
readConfigValues method is started.
readConfigValues method is finished.
```

```
readConfigValues method is started.
readConfigValues method is finished.
Importing and parsing XML file is finished.
loadSource method is started.
Import definition GRCSS is started at Mar 16, 2005 6:47:39 PM.
constantColumnTypes method is started.
constantColumnTypes method is finished.
targetColumnTypes method is started.
Query 'select GRCSID,NAME,ADDRESS,STATUS from  GRCSS'
                              will be executed
targetColumnTypes method is finished.
checkOidLogic method is started.
checkOidLogic method is finished.
querySource method is started.
querySource method is finished.
Query 'select GRCSID, NAME, ADDRESS, STATUS from GRCSS'
                               will be executed
Import block is started.
Working...  1. block of 1
Started from the 1. row.
insertTargetTable method is started.
insertRow method is started.
readSubCounterValue method is started
readSubCounterValue method is finished
Query 'insert into GRCSS(GRCSID, NAME, ADDRESS, STATUS)
                              VALUES (1,'prova','prova','up')' will be executed
insertRow method is finished.
insertTargetTable method is finished.
insertTargetTable method is started.
insertRow method is started.
readSubCounterValue method is started
readSubCounterValue method is finished
Query 'insert into GRCSS(GRCSID, NAME, ADDRESS, STATUS)
                  VALUES (2,'prova2','prova2','down')' will be executed
insertRow method is finished.
insertTargetTable method is finished.
Import definition GRCSS is finished at Mar 16, 2005 6:47:41 PM.
 Duration of importDefinition:2,449 seconds
loadSource method is finished.

Import job No. 4 is started.
readConfigValues method is started.
readConfigValues method is finished.
```

```
Importing and parsing XML file is started.
importSQLStatement  method is started.
importSQLStatement  method is finished.
Importing and parsing XML file is started.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
Importing and parsing XML file is finished.
method executeSQLStatement is started
SQL statement CreateIndexes is started
SQL_stmt : '

CREATE UNIQUE INDEX PKEY ON GRCSS(GRCSID) ;

        ' will be executed
SQL statement CreateIndexes is finished
method executeSQLStatement is finished
 Duration :0,163 seconds

Import job No. 5 is started.
readConfigValues method is started.
readConfigValues method is finished.
Importing and parsing XML file is started.
importSQLStatement  method is started.
importSQLStatement  method is finished.
Importing and parsing XML file is started.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
Importing and parsing XML file is finished.
method executeSQLStatement is started
SQL statement CreatePrimary is started
SQL_stmt : '

ALTER TABLE  GRCSS ADD CONSTRAINT PKEY PRIMARY KEY(ADDRESS,GRCSID) ;

        ' will be executed
SQL statement CreatePrimary is finished
method executeSQLStatement is finished
 Duration :0,252 seconds

Import job No. 6 is started.
readConfigValues method is started.
readConfigValues method is finished.
Importing and parsing XML file is started.
```

```
importSQLStatement  method is started.
importSQLStatement  method is finished.
Importing and parsing XML file is started.
parseImportJDBCParameters method is started.
parseImportJDBCDefaultParameters method is finished.
Importing and parsing XML file is finished.
method executeSQLStatement is started
SQL statement CreateForeigin is started
SQL_stmt : '

        ' will be executed
SQL statement CreateForeigin is finished
method executeSQLStatement is finished
 Duration :0,1 seconds
All rows are commited.
Application is finished.
All jobs duration: 9,220 seconds
```

Appendix B

Oracle Streams scripts

B.1 Set Instantiation SCN

```
CONNECT sys/syspsw@pisatest as sysdba
DECLARE
iscn NUMBER;
BEGIN
iscn := DBMS_FLASHBACK.GET_SYSTEM_CHANGE_NUMBER();
DBMS_APPLY_ADM.SET_TABLE_INSTANTIATION_SCN@PISATEST.CR.CNAF.INFN.IT(
source_object_name => 'SYS.GRCSS',
source_database_name => 'PISATEST.CR.CNAF.INFN.IT',
instantiation_scn => iscn,
apply_database_link => 'RCC1.PI.INFN.IT');
END;
/
```

B.2 Create Streams processes and queue

```
set echo on
spool stream.out

CONNECT sys/syspsw@pisatest as sysdba

EXECUTE DBMS_LOGMNR_D.SET_TABLESPACE('logmntrs');

ALTER TABLE sys.grcss ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
grant all on sys.grcss to strmadmin2;

CONNECT strmadmin2/strmadmin2psw@pisatest
```

```
BEGIN
DBMS_STREAMS_ADM.SET_UP_QUEUE(
   queue_table => 'STREAMS_QUEUE_TABLE',
   queue_name => 'STREAMS_QUEUE',
   queue_user => 'STRMADMIN2');
END;
/
BEGIN
  DBMS_STREAMS_ADM.ADD_TABLE_RULES(
 table_name => 'SYS.GRCSS',
 streams_type => 'CAPTURE',
 streams_name => 'STRMADMIN_CAPTURE',
 queue_name => 'STRMADMIN2.STREAMS_QUEUE',
 include_dml => true,
 include_ddl => true,
 source_database => 'pisatest.cr.cnaf.infn.it');
END;
/
BEGIN
 DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
 table_name => 'SYS.GRCSS');
END;
/
BEGIN
  DBMS_APPLY_ADM.CREATE_APPLY(
      queue_name => 'STRMADMIN2.STREAMS_QUEUE',
      apply_name => 'STRMADMIN_APPLY',
      apply_database_link => 'RCC1.PI.INFN.IT',
      apply_captured => true);
END;
/
BEGIN
 DBMS_STREAMS_ADM.ADD_TABLE_RULES(
   table_name => 'SYS.GRCSS',
   streams_type => 'APPLY',
   streams_name => 'STRMADMIN_APPLY',
   queue_name => 'STRMADMIN2.STREAMS_QUEUE',
   include_dml => true,
   include_ddl => false,
   source_database => 'PISATEST.CR.CNAF.INFN.IT',
   inclusion_rule => true);
```

```
END;
/
CONNECT sys/syspsw@pisatest as sysdba
BEGIN
 DBMS_APPLY_ADM.ALTER_APPLY(
   apply_name => 'STRMADMIN_APPLY',
   apply_user => 'SYS');
END;
/
connect strmadmin2/strmadmin2psw@pisatest
DECLARE
   rs_name  VARCHAR2(64);
BEGIN
  SELECT RULE_SET_OWNER||'.'||RULE_SET_NAME
    INTO rs_name
    FROM DBA_APPLY
    WHERE APPLY_NAME='STRMADMIN_APPLY';
  DBMS_RULE_ADM.GRANT_OBJECT_PRIVILEGE(
    privilege   => SYS.DBMS_RULE_ADM.EXECUTE_ON_RULE_SET,
    object_name => rs_name,
    grantee     => 'SYS');
END;
/
set echo off
spool off
```

B.3 Start the apply process

```
connect strmadmin2/strmadmin2psw@pisatest
BEGIN
 DBMS_APPLY_ADM.SET_PARAMETER(
   apply_name => 'STRMADMIN_APPLY',
   parameter => 'DISABLE_ON_ERROR',
   value => 'N');
END;
/
BEGIN
  DBMS_APPLY_ADM.SET_PARAMETER(
   apply_name => 'STRMADMIN_APPLY',
   parameter => 'parallelism',
   value => '1');
```

```
END;
/
BEGIN
 DBMS_APPLY_ADM.START_APPLY(
   apply_name => 'STRMADMIN_APPLY');
END;
/
```

B.4 Start the capture process

```
connect strmadmin2/strmadmin2psw@pisatest
BEGIN
 DBMS_CAPTURE_ADM.START_CAPTURE(
   capture_name => 'STRMADMIN_CAPTURE');
END;
/
```

Appendix C

C++ application using OCCI for testing Oracle LogMiner

C.1 main.cc

```
#include "occi.h"
#include <fstream.h>
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
using namespace oracle::occi;
using namespace std;


int main() {
    Environment* env = Environment::createEnvironment();
    {
        const string connectString = "";
        const string username = "laura";
        const string password = "laurapsw";
        const string dbname = "pisatest";
        const string table = "laura.grcss";
        unsigned int maxConn = 5;
        unsigned int minConn = 3;
        unsigned int incrConn = 2;
        string c, value, t0="0", t1="0";
        int res;
        ResultSet* res1;
```

```
        StatelessConnectionPool* scp = env->createStatelessConnectionPool (
            username,
            password,
            connectString,
            maxConn,
            minConn,
            incrConn,
            StatelessConnectionPool::HOMOGENEOUS );


        Connection* conn = scp->getConnection();

        Statement* stmt = conn->createStatement();
        c="BEGIN SYS.DBMS_LOGMNR.ADD_LOGFILE(
             LOGFILENAME => '/ORA/dbs03/oradata/pisatest/pisatest/redo01.log',
             options =>SYS.DBMS_LOGMNR.NEW); END;";
        stmt->setSQL(c);
        res=stmt->executeUpdate();
        c="BEGIN SYS.DBMS_LOGMNR.ADD_LOGFILE(
             LOGFILENAME => '/ORA/dbs03/oradata/pisatest/pisatest/redo02.log',
             options => SYS.DBMS_LOGMNR.ADDFILE);END;";
        stmt->setSQL(c);
        res=stmt->executeUpdate();
        c="BEGIN SYS.DBMS_LOGMNR.ADD_LOGFILE(
             LOGFILENAME => '/ORA/dbs03/oradata/pisatest/pisatest/redo03.log',
             options => SYS.DBMS_LOGMNR.ADDFILE); END;";
        stmt->setSQL(c);
        res=stmt->executeUpdate();
        c="alter session SET NLS_DATE_FORMAT ='DD-MON-YYYY HH24:MI:SS'";
        stmt->setSQL(c);
        res=stmt->executeUpdate();
        c="BEGIN SYS.DBMS_LOGMNR.START_LOGMNR(
             options => SYS.DBMS_LOGMNR.DICT_FROM_ONLINE_CATALOG
 +SYS.DBMS_LOGMNR.COMMITTED_DATA_ONLY
+SYS.DBMS_LOGMNR.NO_ROWID_IN_STMT);end;";
        stmt->setSQL(c);
        res=stmt->executeUpdate();
        for(;;)
         {
           stmt = conn->createStatement();
           c="select max(TIMESTAMP) from V$LOGMNR_CONTENTS
                where SEG_OWNER='LAURA' and  seg_name='GRCSS' ";
           stmt->setSQL(c);
```

```
          res1=stmt->executeQuery();
          res1->next();
          t1 = res1->getString(1);
          if (t1!=t0)
           {
             c =t1+".dat";
             ofstream file(c.c_str());
             t0=t1;
             c="select SQL_REDO from V$LOGMNR_CONTENTS
                  where SEG_OWNER='LAURA'
                  and  seg_name='GRCSS' and timestamp>='"+t0+"'";
             stmt->setSQL(c);
             res1=stmt->executeQuery();
             res1->next();
             value = res1->getString(1);
             file<<value<<"\n";
             file.close();
           }
         conn->terminateStatement(stmt);
         sleep(10);
        }
   }
}
```

C.2 perl.pl

```
#!/usr/local/bin/perl
print "Nome del file di input: ";
$file = <STDIN>;
chop($file);
-e $file || die "Il file non esiste!\n\n";
-T $file || die "Il file non e' un file di testo!\n\n";
$si=1;
while($si>0)
{
  print "Stringa da cercare: ";
  $stringa = <STDIN>;
  chop($stringa);
  print "Stringa da sostituire: ";
  $sost = <STDIN>;
  chop($sost);
```

```
  open (IN, "< $file") || die "Impossibile aprire $file.\n\n";
  open (OUT, "> $file.new") || die "Impossibile creare $file.new\n\n";
  while ($r = <IN>) {
     $r =~ s/$stringa/$sost/g;
     print OUT $r;
                              }
  rename ("$file.new","$file");
  close(OUT);
  close(IN);
  print "Vuoi ancora sostituire una stringa? NO(digita 0) SI(digita numero>0)";
  $si=<STDIN>;
  chop($si);
}
```

C.3 LogFWatcher functions

```
void
LogFWatcher::
log_generator()
{
    string cmds = "mysqlbinlog --start-datetime=";
    ostringstream pid;
    pid << getpid();
    realupdfname_ = updpath_ + "/" + updfname_ + "." + intToString(seqn_);
    string tmp = realupdfname_;
    tmp += '.';
    tmp += pid.str();
    ostringstream cmd;
    cmd << cmds << dbmtime_ << ' ' << logfname_ << "> " << tmp;
    system(cmd.str().c_str());
    ifstream tmpf(tmp.c_str());
    seqn_ = (seqn_++)%1000;
    cleanup(tmp, realupdfname_, posn_);
    dbmtime_ = last_dbmtime(tmpf, &posn_);
}


static
void*
LcRCS::
```

```
run(void* arg)
{
    int oldcstate;
    int oldctype;
    int err = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldcstate);
    err = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldctype);
    LogFWatcher* p = static_cast<LogFWatcher*>(arg);
    for (;;) {
        if ((fstat(p->fd_, &(p->sb_)) == -1)) {
            cerr << "LogFWatcher::LogFWatcher(): fd_ == "
                 << p->fd_
                 << endl;
            cerr << "LogFWatcher::LogFWatcher(): CANNOT STAT "
                 << p->logfname_
                 << endl;
        } else {
            time_t mt = p->mtime();
            if (mt > p->lfmtime_) {
                p->lfmtime_ = mt;
                #ifdef BENCHMARK
            Timer queryTime("time");
            unsigned long currentTime = 0;
            currentTime = queryTime.Start();
                #endif
                p->log_generator();
                #ifdef BENCHMARK
            queryTime.Stop();
            cout << "LogFWatcher::Bench: log generated in "
                 << queryTime << endl;
                #endif
                p->save_mtimes();
                p->send_notify();
            }
        }
sleep(p->sleept_);
        pthread_testcancel();
    }
}
```

Bibliography

[1] INFN web site: `http://www.infn.it`, July 2005.

[2] Rick Thompson: *Grid Networking*. Byte and Switch, May 3, 2003.

[3] Joshy Joseph, Craig Fellestein: *Introduction to Grid Computing*. GRID Computing, PRENTICE HALL PTR, IBM press, December 30, 2003.

[4] Ian Foster: *What is the Grid? A Three Point Checklist*. GRIDToday, July 20, 2002.

[5] Ian Foster, Carl Kesselman, Steven Tuecke: *The Anatomy of the Grid. Enabling Scalable Virtual Organizations*. International J. Supercomputer Applications, 15(3), 2001.

[6] Ian Foster, Robert L. Grossman: *Data Integration in a Bandwidth Rich World*. Communications ACM, Volume 46, Issue 11, November, 2003.

[7] Vjayshsnkar Raman, Inderpal Narang, Chris Crone, Laura Haas, Susan Malaika, Tina Mukai, Dan Wolfson, Chaitan Baru: *Data Access and Management Services on Grid*. GGF, DAIS group 2002.

[8] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, Kurt Stockinger: *Data Management in an International Data Grid Project*. Proceedings of the IEEE/ACM International Workshop on Grid Computing, Grid'2000, 2000.

[9] MONARC project web site: `http://monarc.web.cern.ch/MONARC/`, July 2005.

[10] Heinz Stockinger: *Distributed Database Management Systems and the Data Grid*. Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies, April 17-20, 2001.

[11] Dirk Düllman, Wolfgang Hoschek, Javier Jaen-Martinez, Ben Segal, Asad Samar, Heinz Stockinger, Kurt Stockinger: *Models for Replica Synchronisation and Consistency in a Data Grid*. 10th IEEE Symposium High Performance and Distributed Computing (HPDC-10), San Francisco, California, 2001.

[12] Andrea Domenici, Flavia Donno, Gianni Pucciani, Heinz Stockinger, Kurt Stockinger: *Replica consistency in a Data Grid*. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 534, Issues 1-2, 21 November 2004, Special Issue on International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT03), Tsukuba, Japan, December 2003.

[13] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohner, A. Gianoli, K. Lörentey, F. Spataro: *VOMS, an Authorization System for Virtual Organizations*. DataGrid Project, 2003.

[14] K. Lörentey, A. Frohner: *EDG-VOMS-ADMIN Developer's guide*. February 5, 2004.

[15] David Groep & Gridification Team: *VOMS and LCMAPS on Global Permission and Local Credentials*. WP4 meeting EDG, Barcelona, 2003.

[16] Maria Girone: *RLS Production Services*. 10th GridPP meeting, CERN, June 3, 2004.

[17] LCG project web site: `http://cern.ch/lcg/`, July 2005.

[18] INFNGrid project web site: `http://grid.infn.it/`, July 2005.

[19] Constanza project web site: `http://infnforge.cnaf.infn.it/projects/constanza/`, July 2005.

[20] Octopus project web site: `http://octopus.objectweb.org/`, July 2005.

[21] HIT Software web site: `http://www.hitsw.com`, July 2005.

[22] `http://rapidshare.de/files/1124994/DBMoto_4.1.9.zip.html`, July 2005.

[23] **Oracle Database** Administrator's Guide 10*g* Release 1 (10.1). Oracle Corporation, December, 2003.

[24] **Oracle Database** Utilities 10*g* Release 1 (10.1). Oracle Corporation, December, 2003.

[25] **Oracle Streams** Concepts and Administration 10*g* Release 1 (10.1). Oracle Corporation, December 2003.

[26] **Oracle Streams** Replication Administrator's Guide 10*g* Release 1 (10.1). Oracle Corporation, December 2003.

[27] **Oracle Database** Heterogeneous Connectivity Administrator's Guide 10*g* Release 1 (10.1). Oracle Corporation, December 2003.

[28] **Oracle9*i*** Database New Features Release 2 (9.2). Oracle Corporation, October 2002.

[29] `http://www.oracle.com/technology/products/dataint/htdocs/streams_fo.html`, July 2005.

[30] UNIXODBC project web site: `http://www.unixodbc.org`, July 2005.

[31] MySQL web site: `http://www.mysql.org`, July 2005.

[32] **Oracle C++ Call Interface** Programmer's Guide 10*g* Release 1 (10.1). Oracle Corporation, December 2003.

[33] **Oracle Database SQL Reference**10*g* Release 1 (10.1). Oracle Corporation, December 2003.