ISTITUTO NAZIONALE DI FISICA NUCLEARE

Sezione di Pisa

INFN/TC-06/06
1 March 2006

CONStanza: DATA REPLICATION WITH RELAXED CONSISTENCY

Andrea Domenici[1], Flavia Donno[2], Gianni Pucciani[3], Heinz Stockinger[4]

[1] University of Pisa, v. Diotisalvi 2, I-56122 Pisa, Italy
[2] CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland
[3] INFN Pisa, Edificio C, Polo Fibonacci, Via F. Buonarroti 2, I-56127Pisa, Italy
[4] Research Lab for Computational Technologies and Applications, University of Vienna Rathausstrasse 19/9, A-1010 Vienna, Austria

Abstract

Data replication is an important aspects in a Data Grid for increasing fault tolerance and availability. Many Grid replication tools or middleware systems deal with read-only files which implies that replicated data items are always consistent. However, there are several applications that *do* require updates to existing data and the respective replicas. In this article we present a replica consistency service that allows for replica updates in a single-master scenario with lazy update synchronisation. The system allows for updates of (heterogeneous) relational databases, and it is designed to support flat files as well. It keeps remote replicas synchronised and partially ("lazily") consistent. We report on the design and implementation of a novel ``relaxed" replica consistency service and show its usefulness in a typical application use case.

PACS: 89.20.Ff

Published by ***SIS–Pubblicazioni***
Laboratori Nazionali di Frascati

1 Introduction

A Data Grid is a wide area computing infrastructure that provides storage capacity and processing power to applications that handle very large quantities of data. Data Grids rely on data replication to achieve better performance and reliability by storing copies of data sets on different Grid nodes. Grid data management middleware (such as [18]) usually assumes that (i) whole files are the replication unit, and (ii) replicated files are read-only. However, there are requirements for mechanisms that maintain consistency of modifiable replicated data. Furthermore, such data are often highly structured, as in the case of databases, thus making the coarse granularity of file replication impractical if not unfeasible. To address these requirements, we propose a Replica Consistency Service (named *CONStanza*) that is general enough to cater for most types of applications in a Grid environment and meets the more general requirements for Grid middleware, such as performance, scalability, reliability, and security.

The known solutions for the consistency problem in databases [5] are only partially applicable to Grid architectures. Whereas a DBMS has full control of all data it manages, and the data have a homogeneous format, Data Grids usually rely on the underlying file-systems for data manipulation, and the data have widely different formats across different applications and even within each application. File-systems usually do not offer transactions and therefore make consistency harder to maintain. Further considerations on Data Grids and consistency can be found in our earlier work in [13].

Other important differences between Data Grids and distributed databases are the very large number of files a Grid is expected to handle, the dynamic Grid configuration, and the need for scalability.

We present a service *designed* to deal with two basic cases: *file replication* and *database replication* of (heterogeneous) relational database systems. In the former case, the service provides a simple file replacement model when an update is done, but more sophisticated update methods such as the binary difference between two files are possible. In the latter case, the service extracts logs from an updated database and then applies these logs at remote replicas. The system allows for heterogeneous replication since the master database may come of a certain vendor (Oracle or MySQL) whereas the secondary databases may come of another vendor (MySQL). The presented service provides users with the ability to update data using a certain consistency delay parameter in order to adapt to the application requirements. Therefore, we talk about **relaxed consistency** that is fit for applications that do not need fully synchronised data at every given point in time. Furthermore, in the current *implementation* we support only database replication and leave file replication for future work.

In this article, we first review some related work (Section 2). We then expose a basic domain analysis in Section 3. Next, we outline a Replica Consistency Service (RCS) (Section 4) and discuss its high-level architecture (Section 5) that we have prototyped. Finally, in Section 6 we report some preliminary experimental results. A simulation of the architecture implemented in this work was reported in a previous paper [12].

2 Related Work

There is a large number of research papers and development projects focused on either file or database replication, but there does not seem to be any working system that allows for both. In particular, and to the best of our knowledge, there is not much literature about updating databases in a Grid environment. Here, we only provide a short, selected list of products having in mind that many commercial systems exist for database replication.

A project dealing explicitly with file sharing and data consistency in Grids is presented in [3]. This paper proposes to decouple data consistency issues related to performance from those related to fault tolerance. The design of the parts of a consistency service related to performance draws on solutions from the field of Distributed Shared Memory architectures while the parts related to fault-tolerance are designed according to solutions from peer-to-peer systems. Unlike our system it does not provide database features.

A work more closely related to standard Grid architectures is presented in [15], a paper that proposes an update propagation strategy based on reliable multicast communication. The model has been demonstrated in a prototype implementation on a small network. However, for our system multicast is not a viable solution.

A good survey of data consistency models and protocols is found in [21], discussing lazy (similar to our methodology) and only optimistic eager replication approaches.

The problem of maintaining consistency among heterogeneous replicas of databases is addressed in a few commercial products such as **Oracle**, **IBM**, **Sybase**, **Easysoft.com** etc. and several off-the-shelf tools such as **Enhydra Octopus** and **DBMoto**.

Oracle Streams [20] are used to propagate information within a database or from one database system to another. More precisely, a Streams system can capture, stage and manage events in the database (such as changes in its contents or its schema) automatically. Oracle streams are used and work well with Oracle homogeneous databases. The Oracle Heterogeneous Connectivity layer provides a way to update non-Oracle database systems, given one uses proprietary Oracle-MySQL drivers. This is rather impractical in a heterogeneous, open-source Grid environment. This motivates the development of an alternative solution such as ours.

Enhydra Octopus [14] is provided by *ObjectWeb Consortium*, under *LGPL*[1]. This application loads data from a JDBC data source (database) into a JDBC data target, and may perform many transformations defined in an XML file. It supports Oracle to MySQL data transfer. 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.

DBMoto is a product developed by HITSoftware [9]. DBMoto runs on Windows: the entire replication process is configured and managed from a Windows platform. Therefore, such a system cannot be well adapted to the Grid heterogeneous environment. In *Mirroring mode*, DBMoto performs a real time incremental replication based on log management. It does not deal with file consistency.

Another database replication tool is **CORSO** [17] which provides proprietary middleware for distributed applications.

3 Consistency in a Grid environment

In this section we describe the context where a Replica Consistency Service operates and we introduce the main concepts and assumptions. In particular, we describe a typical Data Grid environment.

We will refer to a generic Grid environment with two kinds of Grid nodes: *Computing Elements* (CE) that provide computing power, and *Storage Elements* (SE) that provide storage capacity. Several services compose the *Grid middleware*, providing such functionalities as Grid-wide job scheduling, resource allocation, and data management. The latter class of functionalities includes the *Replica Management Service* (RMS) [8] and the *Replica Consistency Service* (RCS).

The RMS can *replicate* files to optimise access time and availability. The RMS keeps track of the replicas and is able to determine which replica is most convenient for a job running on a given CE. The RMS uses the *Replica Catalogue* (RC; also referred to as Replica Location Service, RLS) [7], a database holding the information about the replicas of each file.

We model a file as an association of its replicas with one *logical file name* (LFN). Replicas are kept on Storage Elements and have the file's contents as an attribute. Another attribute is the name (see Figure 1). We call *physical file name* (PFN) the information needed to access a given replica: this information includes the address of the SE, a pathname within the SE's file system (or some equivalent information), and possibly an access protocol honoured by the SE (e.g., GridFTP).

[1] Lesser GNU General Public License

The property of *strict* consistency for a set of file replicas can be simply defined as follows:

For any pair of replicas of the same logical file, their contents are equal.

In Figure 1, the above definition is expressed (in UML's OCL) as a constraint on the association.

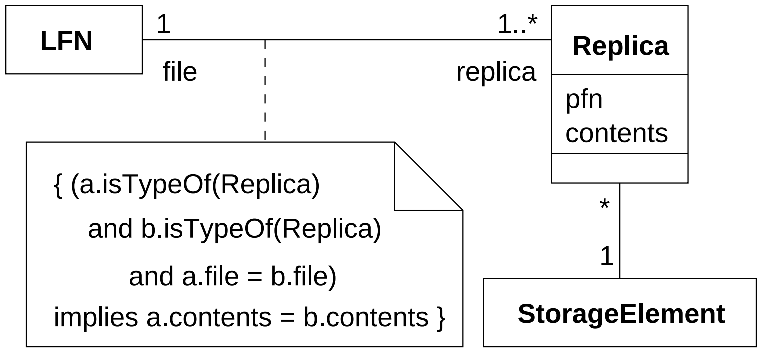

Figure 1: Files and replicas.

The above model must be extended to other kinds of datasets. A dataset may be a *flat* (or *unstructured*) *file*, or a *structured dataset*. Flat files are those whose internal structure is either unknown or ignored for the purposes of replication and consistency, and are accessed with low-level operations (such as byte-oriented `read()` or `write()` calls). Structured datasets are those that can be acted upon by means of high-level operations, such as record-oriented ones. A relational database is a typical example of a structured dataset that is accessed through SQL commands. Databases add another facet to the issue of consistency, since a database with a given schema could be replicated on different systems each having a different implementation. For example, a RC could be replicated to two Grid sites that have database servers from different vendors. These servers would store the same logical information with different physical representations.

To extend the notion of consistency to datasets (including databases), we should then distinguish between the logical contents and their physical representation. We define the *semantic function* $\sigma_{d,i}$ *of the i-th replica of dataset d* as the mapping of the physical representation constituting the replica to its logical contents. For flat files, the semantic function is the identity function, and for databases it is the mechanism that extracts information from the database. Then, replicas i and j, containing the representations r_i and r_j, respectively, of a dataset d, are consistent iff $\sigma_{d,i}(r_i) = \sigma_{d,j}(r_j)$.

The above definition does not take physical constraints into account. Such constraints arise from the large-scale distributed structure of the Grid, where latencies may be high and communication failures rather frequent. Therefore, different replica synchronization policies exist, each dealing differently with the constraints.

In our system it is important to distinguish between the following types of replicas since they have different semantics and access permissions for end-users:

- **Master replica**, i.e. one replica that can be updated by end-users of the system. The master replica is the one that is by definition always up-to-date.
- **Secondary replica** (also referred to as secondary copy) is a replica that is to be updated/synchronised by CONStanza with a certain delay to eventually have the same contents as the master replica. Obviously, the longer the update propagation delay is, the more un-synchronised are the master and the secondary replica, and the bigger is the probability to experience stale reads on secondary replicas. Furthermore, end-users only have read-access to secondary copies since updates are only made by CONStanza to avoid conflicts in the update process.

Use Cases

In order to prove the usefulness of our system, we applied it to several real-world use cases from the LCG [19] and EGEE [10] projects. One use case deals with updates of user group information used by the Virtual Organisation Membership Service (VOMS) [2]. The service stores several kinds of user authentication information in a relational database which is currently a single, central database.

For fault tolerance and load balancing reasons the application server and its database are replicated to a few sites (in the order of 10). The database backends are heterogeneous (i.e. from different vendors), the master being either Oracle or MySQL. An important feature is the High Energy Physics computing model (that is used in both LCG and EGEE): a hierarchical, multi-tier architecture is used for the distribution of computing and storage resources where lower tiers often use open source systems such as MySQL and higher tiers use Oracle. Data is then partially replicated mainly from higher to lower tiers, meaning that only *some* master tables are replicated and need to be synchronised. As of writing there are more than 150 sites (belonging to different tiers in the distributed architecture) with several Grid services requiring VOMS authentication. Each of these services accesses the VOMS server every 6 hours. New user information is inserted into the master database at a frequency of about once a day per virtual organisation. Consequently, the update frequency is rather low and can easily be managed by the proposed system.

In case of LCG's file catalogue (LFC [4]), which is currently centralised with a single database backend, we have seen that during periods of heavy usage updates to the database occurred at a frequency of once in a few minutes. Even if in this use case the update frequency is higher, the user tolerates an update delay for secondary replicas in the order of one hour. The number of foreseen LFC replicas is about 5 to 10, i.e. well within the limits of our system.

4 Architecture

An application (including other middleware components) sees the dataset as a single logical entity identified by a *Logical Dataset Id (LId)* (an LFN being a special case of LId). When it needs to modify the dataset, it accesses one of the replicas, it modifies its contents, and then tells the RCS to propagate the change to all the other replicas. Therefore, we distinguish the following steps in the overall replication system:

- **Update dataset**: the process of modifying a single replica.
- **Update propagation (UP)**: the process of reproducing the modification on the set of all replicas.
- **Update replica process (URP)**: this includes all steps required to identify which changes have been done on a given dataset (master replica) and then apply the update propagation process to all existing replicas.

We can distinguish between *synchronous* and *asynchronous* propagation [16,1]. A synchronous propagation policy ensures that no replicas are available until they all have been updated. An asynchronous propagation policy allows replicas with old contents to be accessible while others are being updated. This means that the consistency requirement is violated for the time necessary to complete update propagation, but this relaxed policy is more realistic in a Grid environment. However, there may be situations where synchronous propagation is required. Therefore, the RCS design allows for adding different update propagation policies.

Currently, we have implemented the asynchronous update protocol which uses a *single master* approach, where only one replica can be modified by the users. This is a standard solution that can be found in many commercial products such as Oracle, MS SQL-Server etc.

4.1 Main Components

In the proposed design, the following basic components provide the required functionalities:

Global Replica Consistency Service (GRCS): the main service front-end toward applications. In other words, the GRCS represents the main entry point for update requests whereas the system itself consists of several other distributed components (see below).
Local Replica Consistency Service (LRCS): it is responsible for the registration of new local replicas that it manages directly as well as the update of these replicas. Internally, each of these local services has a catalogue to manage specific metadata that are required to update the local dataset.
Replica Consistency Catalogue (RCC): to store the metadata used by the RCS, such as location and status of each replica as well as LRCS information.

The interaction of the above components is shown in Figure 2. These components are general enough to be applied to various types of datasets (i.e., flat files or database files) which require additional components that take care of the actual update operations.

The basic interface to the RCS for an end-user is a command line client that sends *update* requests to the Replica Consistency Service. In more detail, end users can *modify* only master replicas (controlled by a Master LRCS) whereas secondary copies can only be accessed in *read* mode. That is the classical master-secondary approach that allows for fast read access (taken into account that reading from secondary copies can result in reading non-synchronised, out-of-date contents, i.e stale reads), and limited and centralised write access. There are certainly other, more complicated replication scenarios with several masters for a given set of replicas but in our approach we keep the number of transactions between services minimal in order to limit the complexity of the system. Experience shows that overly complex Grid middleware has sometimes failed to be deployed for production service.

In the most basic scenario, a user first needs to update the master replica and then issues the `update` client command to have the RCS replicate the change. The system also assumes that the data are already replicated to a few sites and their locations are stored in the replica consistency catalogue. We use a subscription model to achieve this. In particular, LRCSs subscribe with the GRCS in order to retrieve updates on local, secondary replicas.

We now discuss replication for files and databases, assuming a single-master policy, where a single LRCS holds the master files that can be updated by end-users.

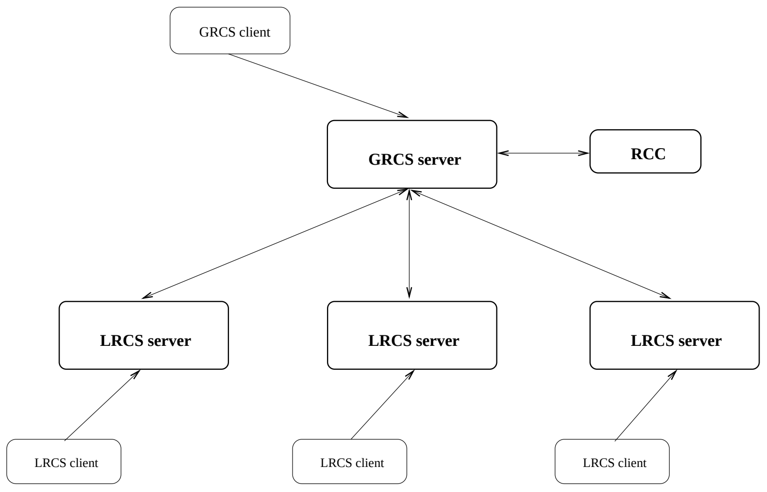

Figure 2: Architecture of the consistency service.

4.2 File Replication

4.3 File Replication

The *master* LRCS module (i.e. the service that sits on a Storage Element where updates are allowed for end users) receives and executes update requests for the files under its control. The difficult part is here how to *lock* the master and the secondary files that reside in the storage system since conventional file systems only provide advisory locks. Therefore, the preferred operation model is to use a storage system that provides such features and does not allow users to delete files while another one writes them. Grid storage systems such as Castor [6], dCache [11] etc. have such features.

For the update of the secondary replicas, there are two basic scenarios. One one hand, one can use a binary difference tool that detects the file changes and then applies them. On the other hand, in some cases where the file is small and the update is comparably big, the binary update is not efficient and one can simply replace the entire file.

For the file replication use case, the update propagation process starts immediately after a master file has been modified and the end user has called the client program `rcs-update-file`. This is in contrast to the database use case (see below) where a Log Watcher component is used to collect certain updates which are then propagated at certain intervals. Such an approach would also be possible for files (the GRCS would need to collect all updates on individual files) but is a bit of an overhead since there is no central log creator/collector for files residing in a file system.

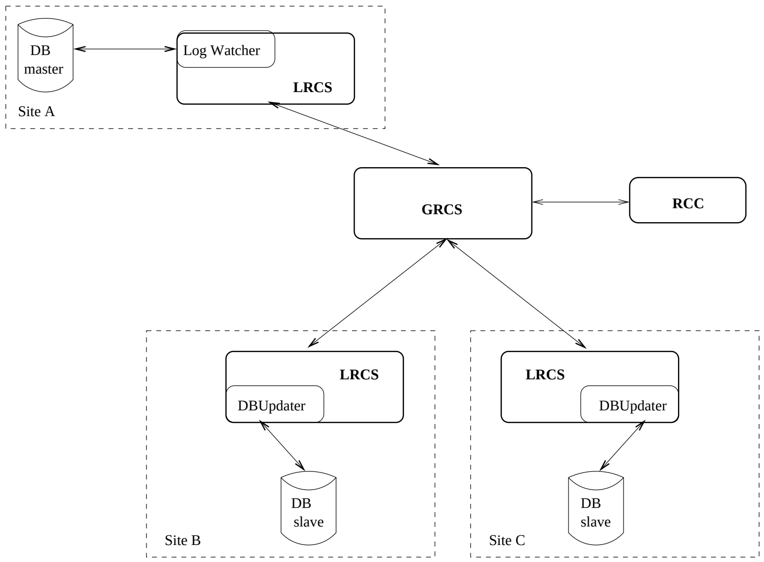

Figure 3: Architecture of the consistency service.

4.4 Database Replication

For the database scenario we automated the update step by adding a **Log Watcher** component to the LRCS (see Figure 3). It *periodically* checks if changes have been made to the master database, it extracts the last update statement from the log and prepares an update unit (also referred to as "log file") to be sent to the GRCS with *update* command. Afterwards, LRCSs holding secondary replicas retrieve the update unit and apply it locally to the secondary replica to make it consistent with the master replica. In this way, the RCS can be set up to synchronise databases at predefined intervals and does not require any user interaction. In other words, the update delay (and therefore the relaxation of the consistency between replicas) can be configured, depending on the needs of the applications.

In detail, all updates are done at the master database that keeps track of all the changes done on the data. At a certain point in time (i.e. based on the configurable interval with which the Log Watcher checks the contents of the master database) these changes are recorded and stored in a database log file. In other words, we do *not* support synchronous replication, but only propagate updates that have been collected and logged over a certain time that can be defined by the service administrator.

The longer the time between two log generation events, the more likely it is to have inconsistent secondary replicas. The details of log generation depend on the database management system and are handled by the **Log Watcher** component which includes the log generation functionality.

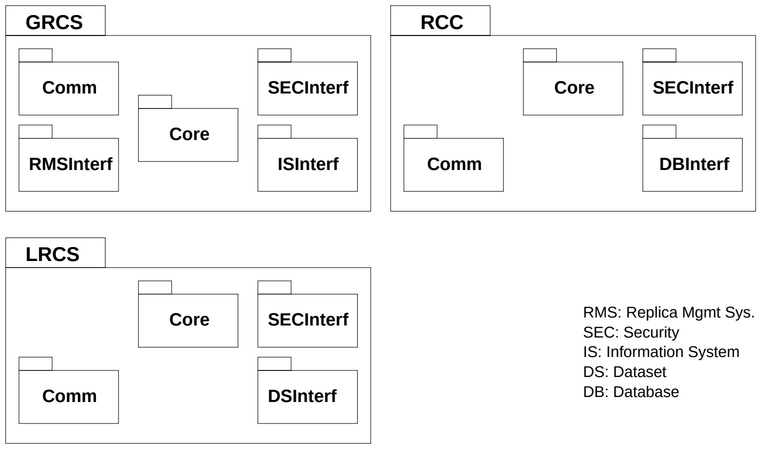

Figure 4: Subsystems of the consistency service.

Once the database log (a list of SQL statements) has been generated, it is retrieved by LRCSs holding secondary copies of the data (DB secondary replica in Figure 3) and then applied to all the secondary replicas. This is done by the **DBUpdater** component. It is a specialised component to support different vendors.

5 Design and Implementation

An advanced prototype for the RCS has been designed and implemented in C++ (gcc 3.2.x on GNU/Linux) and gSOAP 2.7 for the communication between the components of the RCS. The design allows for replication of flat files and relational databases. Currently, we have mainly implemented the database update scenario due to its high priority for our nearest user communities. However, the basic building blocks for the file replication scenario are there and can easily be extended to have a full system that also replicates files.

The RCS has a Web Service interface, but its modular structure allows for alternative solutions to be used if needed (i.e. other communications systems and/or protocols). Figure 4 shows the main subsystems of the service.

Each subsystem has a **Core** package implementing the subsystem functionality, and a **Comm** package supporting communication among the subsystems and between the RCS and its clients. Other packages interface the subsystem to other internal or external components: to the Replica Management System, to the Grid security mechanism, to the Grid information and monitoring system, to the database used by the RCC, and to the datasets.

The **DSInterf** module applies updates to the different kinds of datasets. In the case of flat files, this module interacts with the Storage Elements and it must be general enough to work with the different SE interfaces used or being developed in different Data Grid projects.

In the case of structured datasets, it must be able to update *heterogeneous DB replicas*, i.e., DBs that share the same logical schema but have different physical representations. Most DBMSs provide means to extract logs of updates.

The Log Watcher component currently has two specialised versions to deal with both MySQL and Oracle databases. Using the LRCS client interface we can instruct an LRCS server to monitor a database specifying the vendor type (e.g MySQL or Oracle). If we choose the MySQL type, the Log Watcher will monitor the database through the *mysqlbinlog* utility. In case of an Oracle type instead, the Log Watcher will make use of the Oracle *Log Miner* package to view the Oracle on-line redo logs[2]. When a log is extracted from an Oracle database to be sent to MySQL replicas, a *translation process* is needed to resolve some SQL difference in the dialect used by the two database systems. In our RCS this process is automatically done inside the Log Watcher component with a module that uses Flex and Bison to parse/translate the log file. Currently, the SQL translation is general enough to support our specific use cases.

The RCS can then send a log to the LRCSs responsible for the DB replicas, where the **DSInterface** will apply them through a **DBUpdater** module.

Fault Tolerance

Most of the problems in implementing a reliable consistency service, in the case of a fairly simple protocol like the asynchronous single master, come from the unreliability of the connections among RCS's internal and external components, that is a main concern in a highly dynamic environment such as a Data Grid. Since the GRCS and the LRCSs can be geographically distributed, their connections are subject to failures. Also the link between an LRCS and its database replica can be temporarily unavailable. It is mainly built around a queue that stores undelivered messages when an endpoint is unreachable.

This system is flexible enough to be used with both push and pull based algorithms. In a push method it is up to the LRCS that comes up to request its pending updates while in a pull scenario it is the GRCS that periodically checks if the LRCS is again available. Currently, we have an LRCS client call to execute the push method.

Support for a quorum system [22] is also provided, giving the client the possibility to execute an operation only when a specified number of replicas are available.

In our design the GRCS plays a main role. It is very important for fault tolerance

[2]http://oraclesvca2.oracle.com/docs/cd/B14117_01/server.101/b10825/toc.htm

and load balancing reasons to be able to run more than one GRCS at the same time. A few GRCSs can work in parallel accessing the same RCC.

As regards the RCC, it might represent a single point of failure. In order to avoid this we replicate it using its native solutions, such as MySQL replication. In particular, the RCS knows several locations of the RCC database and connects to an alternative, replicated RCC database if the original database is not available.

6 Experimental Results

In this section we provide several experimental results on wide-area replication of relational databases. We focus on showing the functionality and some performance aspects of the system in order to encourage further development and allow for an analysis of its limitations. In order to make a first evaluation of CONStanza we concentrated on issues such as configurability, load balancing and fault tolerance.

We ran a set of tests demonstrating the functionality of replication between a MySQL or Oracle master and MySQL secondary databases. For MySQL-to-MySQL replication, each database is co-located with an LRCS (master or secondary), i.e. the CONStanza service and the database management system run on the same machine. The main CONStanza service (GRCS) is also located at the site (machine) where the master database is located. For the Oracle-to-MySQL case, the database server is located on a separte machine.

In the experimental setup we used several machines on 4 different sites (INFN Pisa - Italy, University of Pisa, INFN Bologna - Italy, University of Vienna - Austria) connected via wide-area network links. All machines have heterogeneous, standard off-the-shelf hardware devices (mainly Pentium IV with either 1 or 2 processors, between 1.7 and 2 GHz, 512 MB RAM, with FastEthernet network cards). The Oracle server is hosted on a machine with 4 CPUs Intel Xeon 2.20 GHz, 2 GB RAM. All our machines run the *GNU/Linux* operating system (either RedHat Linux 7.3 or Scientific Linux CERN 3.0.x), a standard environment for scientific applications. In order to demonstrate that CONStanza can deal with hardware bottlenecks, we also included a rather old and slow machine (Pentium III, 450 MHz, 256 KB RAM and a slow network card). The database management systems we used are MySQL 4.0.25 and Oracle 10g Enterprise Edition Release 10.1.0.2.0.

6.1 MySQL-to-MySQL Replication

In the first set of tests we measure the end-to-end performance of an update process of MySQL databases using CONStanza.

The tests have been set up as follows:

- The MySQL **master database** is replicated to several machines and sites. It is continuously updated by clients.
- The **Master LRCS** is located on the same machine as the master database, with the Log Watcher checking the changes of the MySQL DB.
- Additionally, the **GRCS** - acting as the main entry point for clients of the system - is located on the same machine.
- We dynamically add and remove several **Secondary LRCSs** on the same LAN as well as at remote sites connected via WAN links with the round trip times (RTT) ranging from 0.1 ms to 45 ms.

The Log Watcher at the master DB is configurable in order to change the time interval with which logs are recorded. The GRCS then synchronises the secondary LRCSs with that time interval. This is the main configurable parameter that can be used to relax the consistency between the master and the secondary database. We refer to this as the *period t(Lg)* of the Log Watcher that is expressed in seconds.

The number of operations for each update is related to the period $t(Lg)$ and to the frequency of the operations issued by the database users. We started with a short period of 5 seconds and analysed the overall performance of the update replication process. In our tests we inserted 100, 1000, 10000 and 100000 entries into a table of a MySQL database and measured the update time. In our tests, an entry consists of 6 attributes (of type double). This was randomly chosen since neiter the data-type nor the number of attributes per value has significant consequences for the overall tests performed. The inserts took from 30 ms (100 inserts) to 11 min (100000 inserts) as indicated in Figure 5.

The entire update replica process (URP, introduced in Section 4) consists of the following steps with performance figures shown in Figures 5 and 6:

1. **Log generation**: Creation of an **log file** (update unit) for a new log with the Master LRCS (note that the log itself has been made available by the MySQL log creator that is constantly monitoring activities done on the MySQL database). A log file is created that only contains the changes made to the database since the last log generation. The **size of the log file** depends on the number of inserts made during $t(Lg)$. In our tests, the log generation time scales almost linearly from 180 ms (100 inserts) to 151 s (100000 inserts). The size of the log file scales in the same way (27 KB, 270 KB, 2.6 MB and 26 MB, respectively).
2. **Replica location and notification**: This step consists of looking up the replicas in the RCC for a given master database. Next, notification messages are sent to

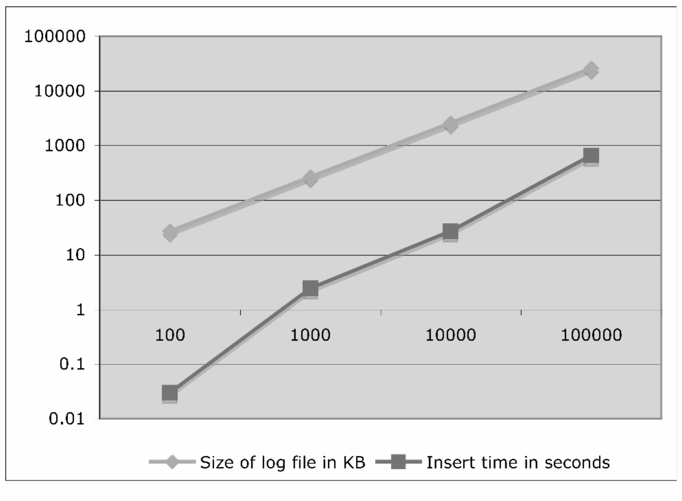

Figure 5: Time to insert values into the MySQL master database vs. size of the log generated for 100 to 100 000 inserts.

LRCSs to update the secondary copies. Additionally, the acknowledgments from the LRCSs are taken. Given the number of 3 secondary replicas and the WAN latencies (RTTs of maximum 45 ms), this step takes about 100–200 ms.

3. **Log Transfer**: Transfer of the log file from LRCS Master to LRCS Secondary via GridFTP. The transfer times varied between the different LRCSs since one is on the same LAN as the GRCS and the other LRCSs are on remote sites connected via WANs. For each file transfer a GSI handshake is required (a constant time independent of the transfer size). The overall transfer time for the log files varied between 1 s (27 KB log file transfer on a LAN) and 9 s (26 MB file transfer on a WAN). Given the increasing number of inserts in the period $t(Lg)$, the file size has a direct relation to $t(Lg)$.

4. **Database Update**: The received log file is applied by the DBUpdater in order to update the secondary databases. Again, the time it takes to apply the entire log is directly proportional to the size of the log file. In our tests it was in the range of 100 ms (100 entries) to 102 s (100000 entries) for the Pentium IV based LRCSs. The Pentium III based LRCS had a severe performance problem due to small RAM and slow processor, and the inserts ranged from 3.6 s (100 entries) to 360 s (for 10000 entries). We did not include this machine in the run with 100000 entries since the update took too long compared to the other machines.

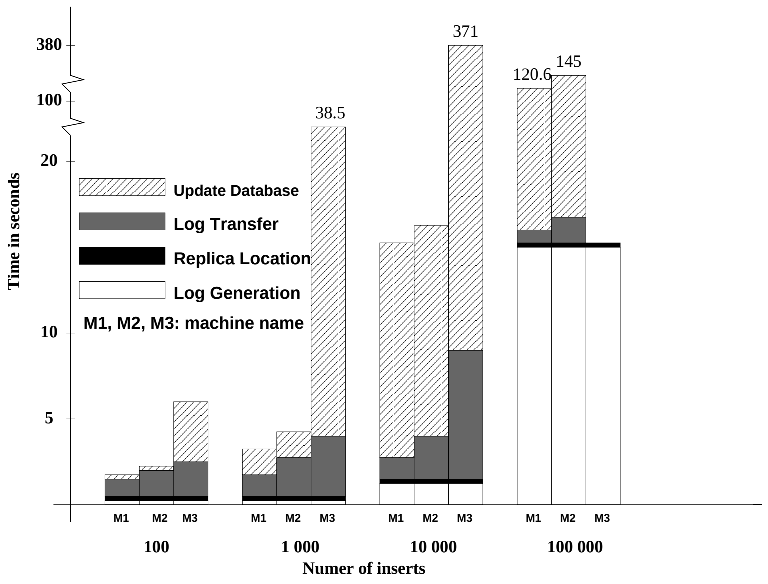

Figure 6: Overall performance parameters in seconds for the update replica process for 3 LRCSs (M1, M2, M3) using 100 to 100 000 database inserts. Note the different scaling on the y-axis or values greater than 20.

Consequently, the overall time seen by the GRCS for the entire update process of all replicas depends on the single steps stated above. Note the GRCS is notifying all the LRCSs in parallel (via parallel threads) which means that performance bottlenecks mainly depend on the speed of the network and the slowest database operations. In our case, the Pentium III was the performance bottleneck. If we only look at the two “fast” LRCSs, the entire update process lasted between 2 s (for 100 entries) and 120 - 145 s (for 100000 entries). We also needed to adapt the log checking period $t(Lg)$ from 5 s (for 100 and 1000 entries) to 60 s (for 10000 entries) to 800 s since the time to create the batch insert varied from 30 ms to 11 min.

The tests have shown that the system is fully functional and performs well for different update scenarios that are all within the requirements specified in the Use Cases in Section 3.

6.2 Fault Tolerance

As a next step we tested the fault tolerance of the system and added also more LRCSs. Given our master and secondary LRCSs, we now assume that one LRCS has crashed or the network connection is lost. Shortly afterward updates have been made to the LRCS master. However, the secondary LRCS gets no information about the changes made in the master database. In this case, the GRCS keeps track of all the updates that were not correctly applied to the LRCSs. In particular, the GRCS keeps all log files locally in order to allow for a *fully serialised* recovery of sites that did not receive the latest logs. That is an advantage of a single master approach since all updates are maintained centrally.

Since each LRCS keeps a local database with references to its local replicas and to its GRCS, the LRCS can easily recover its state. When the LRCS is up again, the command line tool `rcs-complete-op` needs to be called, which then synchronises all the missing updates. This can also be fully automated by running a cron job that periodically checks if the LRCS is still available and calls the recovery process if needed.

In our fault tolerance tests with 4 LRCSs we removed 1, 2, 3 and finally all 4 LRCSs and then reconnected them again for recovery. The GRCS keeps track of all these failures and therefore also reports how many replicas have been correctly updated for each update request. The tests have proven that the system is fault tolerant for this type of LRCS failures.

6.3 Orcale-to-MySQL Replication

In the following tests we show heterogeneous replication where the master database is hosted in Oracle and secondary replicas in MySQL databases.

To test the heterogeneous replication we used an Oracle server located in Bologna and two machines located at INFN-Pisa (pcgridtest2 and pcgridtest3). On pcgridtest2 we ran the GRCS and the LRCS master with the Log Watcher component that remotely monitored the Oracle database in Bologna. On pcgridtest3 we placed another LRCS to manage the MySQL secondary replica. Note that for Oracle-to-MySQL replication the difference with respect to the MySQL-to-MySQL replication is only the Log Watcher component that monitors the master database. Thus, we focused our tests on this component.

The secondary replica has been created with the same schema as the Oracle database, i.e. a table with 3 integer columns and 3 VARCHAR(40) columns. The Log Watcher has been configured to watch the master database every 5 seconds. Then we updated the master database with batches of inserts issued every 10 seconds. We repeated the test three times using different batch sizes: 20, 100, 200 and 1000 insert statements. We measured the time needed by the Log Watcher to create the update unit and to translate the SQL statements.

1. Log Generation: this is the same measurement used for the MySQL-to-MySQL replication with the difference that now the log generation phase involves a remote query to the Oracle database. For this reason the measured time is greater with respect to the MySQL-to-MySQL case, and its average values vary from 3.1 s (for 20 inserts) to 14 s (for 1000 inserts).

2. Log Translation: this measurement includes the time needed by the Log Watcher to translate the SQL statement. This process is part of the Log Generation phase and so this time interval is already included in the previous measurement. Its impact however is minimal since its average has been proven to vary from 1 ms (for 20 inserts) to 56 ms (for 1000 inserts).

The logs are then transferred to the LRCS and applied to the MySQL database in the same way as shown in the previous section. In summary, the Oracle-to-MySQL replication process is more time consuming than the MySQL case, but the increased added value for this functionality is very important for the end-users that allow for certain delays in the replica update process.

7 Conclusions

Data Grids, with their large scale, heterogeneity of node architecture, variety of application requirements and data formats, and autonomy of sites, are a new setting for the long-studied problem of replica consistency. In this paper we have proposed a Replica Consistency Service conceived for Data Grids, and outlined its main requirements and its high-level architecture. A prototype of this service is operational and is going to be deployed both as a production service and as a means to evaluate different propagation protocols.

The system allows for a dynamic way of asynchronously updating replicas by changing the frequency of the checks for database modifications (inverse of the period $t(Lg)$). Since we do not aim at a fully synchronous replications system (see problems in [16]), our current prototype implementation satisfies the basic requirement for relaxed update synchronisation.

The experimental results have shown that the system is functional for replicating MySQL databases as well as contents of Oracle to MySQL systems. The latter is a much needed feature that addresses the problem of heterogeneous database replication in a Grid environment. Since the CONStanza design also provides for file replication and the necessary stubs have been prepared, it can be considered as a general solution to simple update synchronisation where data do not need to be fully synchronised all the time.

Acknowledgments

We thank Enrico Galliero for providing test scripts for the performance tests with MySQL. We also thank Laura Iannone for her help with Oracle Streams and some tests with related database management systems. Thanks to Kurt Stockinger for valuable comments.

References

[1] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. VLDB Conference (Trodheim, Norway, Sep. 2005).

[2] R. Alfieri et al. Managing Dynamic User Communities in a Grid of Autonomous Resources. CHEP 2003 Conference (La Jolla, California, March 24-28, 2003).

[3] G. Antoniu JFDeverge, S. Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. Rapport de recherche de l'INRIA Rennes (RR-5467) (Jan. 2005).

[4] J. Baud, J. Casey, S. Lemaitre, C. Nicholson. Performance Analysis of a File Catalog for the LHC Computing Grid. 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14) (Research Triangle Park, North Carolina, July 24-27, 2005).

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, A. Silberschatz. Update Propagation Protocols For Replicated Databases. ACM SIGMOD International Conference on Management of Data (Philadelphia, Pennsylvania, June 1-3, 1999).

[6] CASTOR storage manager, http://cern.ch/castor, May 2005.

[7] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Scharzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. International ACM/IEEE Supercomputing Conference (SC 2002), IEEE Computer Society Press, Baltimore, Maryland, November 16-22 (2002).

[8] D. Cameron, J. Casey, L. Guy, P. Kunszt, S. Lemaitre, G. McCance, H. Stockinger, K. Stockinger, G. Andronico, W. Bell, I. Ben-Akiva, D. Bosio, R. Chytracek, A. Domenici, F. Donno, W. Hoschek, E. Laure, L. Lucio, P. Millar, L. Salconi, B. Segal, M. Silander. Replica Management in the EU DataGrid Project, International Journal of Grid Computing, Springer Science+Business Media B.V., Formerly Kluwer Academic Publishers B.V., 2(4):341-351 (2004).

[9] DBMoto: http://www.hitsw.com (April 2005).

[10] EGEE Project web site: http://www.eu-egee.org/ (June 2005).

[11] dCache storage manager, http://www.dcache.org/ (May 2005).

[12] A. Domenici, F. Donno, G. Pucciani, H. Stockinger, K. Stockinger. Replica Consistency in a Data Grid. IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT03) (Tsukuba, Japan, Dec. 1-5, 2003).

[13] D. Düllmann, W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, K. Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid, *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)* (San Francisco, California, August 7-9, 2001).

[14] Enhydra Octopus: http://octopus.objectweb.org (April 2005).

[15] C. Gaibisso, F. Lombardi. A Reliable Multicast Approach to Replica Management for Grids. *23rd IASTED Int. Multi-Conference Parallel and Distributed Computing and Networks* (Innsbruck, Austria, Feb. 15-17, 2005).

[16] J. Gray, P. Helland, P. E. O'Neil, D. Shasha. The Dangers of Replication and a Solution. *SIGMOD Conference* (Tucson, AZ, May 12-14, 1996).

[17] E. Kühn. The Zero-Delay Data Warehouse: Mobilizing Heterogeneous Databases. *29th VLDB Conference* (Berlin, Germany, 2003).

[18] P. Kunszt, E. Laure, H. Stockinger, K. Stockinger. Advanced Replica Management with Reptor. In *5th Int. Conf. on Parallel Processing and Applied Mathematics* (Czestochowa, Poland, September 7-10, 2003).

[19] LCG project web site: http://cern.ch/lcg/ (June 2005).

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

[21] Y. Saito,M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 5(3):1-44 (2005).

[22] A. Wool. Quorum Systems in Replicated Databases: Science or Fiction? Bulletin of the IEEE Computer Society Technical Committee on Data Engineering (1998).