5. API IMPLEMENTATION ON DESKTOP SYSTEMS

The API implementation for Desktop systems is based upon Windows 7 Operating System and .NET Framework 4.0, and it works on both 32-bit and 64-bit OS versions.

The Local Storage Manager component used is the Windows NTFS File System and the Client API calls it *via* the System.IO .NET library.

The Cloud Storage Manager component is the Windows Azure Storage, and the Client API calls it via the WindowsAzure.Storage.Client library.

In the next sections we describe how these technologies are used to realize the main mechanisms that are part of this proposed abstraction.

5.1. Types, persistence and access methods

5.1.1. Main index and Main index file

Main Index description – The main index is implemented using a .NET 4.0 Dictionary provided by the .NET Framework 4.0 System.Generics.Collections library. Below follows the Main Index definition:

```
internal Dictionary<string, MainIndexRecord>;

internal class MainIndexRecord
{
  internal string recType;
  internal DataFileToken token;
}
public class DataFileToken
{
  public long offset;
  public long length;
}
```

Each Dictionary entry key is a string representing a filename (e.g. “Readme.txt”), and for each filename it is stored, its value contains:

- its Blob's type (e.g. "PUC_Rio.Song");
- Its DataFileToken containing its position and length in the Metadata file.

When an application finishes the CDB operations by issuing the **CDB.Close()** instance method, the Dictionary contents are persisted to a binary file in the File System. When an application creates a CDB instance calling its constructor, if an index file exists in the file system it is read into the Dictionary in memory; otherwise an empty index file is created and the Dictionary starts empty as well.

The main Index File structure follows:

Name convention	CDB_*DatabaseName*.IND			
File Type	Binary, records wth varied size			
File sample				
"Brahms01.m4a"	"Song"	271	143	
"Brahms02.m4a"	"Song"	271	414	
"Brahms09.m4a"	"Song"	0	-1	*(stored in the Cloud)*
"UML01.wmv"	"Video"	293	143	
"UML01.wmv"	"Video"	302	436	
"UML02.wmv"	"Video"	0	-1	*(stored in the Cloud)*

Table 1: Main Index File Structure

Main Index Consistency and Recovery – As it was previously explained, the Main Index is an in-memory structure that is serialized only when the CDB is closed. We must therefore ensure that in case of a system crash we don't lose all entries added to it, nor lose track of deleted entries. We use an Operations Log to track Main Index changes, and we use it to recover the Main Index from system crashes.

During the CDB instance initialization, it searches for an Operations Log:

- If it doesn't exist, a new one is created. Each operation that writes to the Main Index is recorded in this log, which is stored in the Local container, as a non-buffered operation. If the CDB is closed properly, the whole Main Index will be serialized to the Local Container and this Operations Log will be erased. If the system halts

before proper closing, the Main Index won't have a chance to be serialized, but the Operations log will remain and will have stored all operations done on that session, and the next time the CDB is initialized it will find this operations Log.

- If it exists, it will first load the content of the Main Index file to a Dictionary in memory. Then it will use the Operations log to update the Main Index by applying the changes made during the last session, which had prematurely been halted.

The Operations Log structure follows:

Name convention	CDB_*DatabaseName*.SAV1			
File Type	Text, records wth varied size			
File sample				
"Insert"	"Brahms_01.m4a"	"Song"	"143"	"271"
"Insert"	"Brahms_02.m4a"	"Song"	"414"	"271"

Table 2: Operations Log

5.1.2.Metadata and Metadata File

Each blob that is added to the local container may have metadata associated to it (e.g. "Script.Author", "Report.Company" etc.) and this metadata is stored in a metadata file specific to its entity class. However, if a file was stored in the Cloud, its metadata is stored the Cloud as well and will not show up in any metadata file. The reason behind this implementation choice is simple – if there's no space for storing a file locally, it is likely that there will be no space to store its metadata locally, either.

For performance reasons, one application that defines N blob-derived Entity classes will use N metadata files.

The Metadata File structure follows:

Name convention	*DatabaseName_DataTypeName* .RDF
File Type	RDF/XML
File sample	
<?xml version="1.0" encoding="utf-8"?>	
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:cdb="http://www.puc-rio.br/">	
<rdf:Description rdf:about="file:///C:/R2_January_2012/CDB/Container/Brahms_01.m4a">	
<cdb:size>1000</cdb:size>	
<cdb:author>SongAuthor</cdb:author>	
<cdb:title>SongBrahms_01.m4a</cdb:title>	
<cdb:year>2001</cdb:year>	
</rdf:Description>	
<rdf:Description rdf:about="file:///C:/R2_January_2012/CDB/Container/Brahms_02.m4a">	
<cdb:size>2000</cdb:size>	
<cdb:author>SongAuthor</cdb:author>	
<cdb:title>SongBrahms_02.m4a</cdb:title>	
<cdb:year>2002</cdb:year>	
</rdf:Description>	
</rdf:RDF>	

Table 3: Metadata file

*** RDF/XML representation** – Metadata files are text files. Records are stored using the RDF/XML[25] standard. Each record stores the physical location of a blob and its related metadata. If the Entity Classes are created according to known ontologies, this representation will make it easier for the Client Application to publish the database content across the Internet.

5.2. Consistency and Rollback

CDB operations of Adding, Updating or Deleting blobs need to be performed in a way that they do not leave the system inconsistent. Maintaining a Rollback Log is the technique that will guarantee atomicity for these operations, and it was already explained in section 4.1.2.3. From an implementation viewpoint, an in-memory XML tree fits well the needs for tracking the rollback system: One complex operation (add, delete or update a blob) may contain multiple small operations that need to be tracked together, so we log one **transaction** for each of these operation calls. Moreover, each operation may require a different set of parameters to be logged. These requirements naturally suggest a hierarchy, and that's XML at its essence.

The Rollback File Structure follows:

Name convention	CDB_*DatabaseName* .SAV0
File Type	XML
File sample	

```
<?xml version="1.0" encoding="utf-8"?>
<OGF03a>
 <transaction number="1002897798">
  <invoke name="Rollback_AddBlobToMainIndex" target="Brahms_03.m4a" />
  <invoke name="Rollback_WriteDataToDataFile" target="Song">
   <params>
    <param value="file:///C:/R2_January_2012/CDB/Container/Brahms_03.m4a" />
   </params>
  </invoke>
  <invoke name="Rollback_AddFileToLocalContainer" target="PUC_Rio.CDB">
   <params>
    <param value="C:\R2_January_2012\CDB\Container\Brahms_03.m4a" />
   </params>
  </invoke>
 </transaction>
</OGF03a>
```

Table 4: Rollback file

5.3.Database limitations

Currently this implementation does not provide any specific database Authentication/authorization mechanism. For Local storage, it is up to a Systems Administrator to grant or deny access to users using the File System capabilities; the CDB itself does not lock the Local Container for specific users. For Cloud Storage, a very basic mechanism is provided by the Azure CloudStorage API. Each Request to Cloud resources issued by a CDB instance will contain an encrypted header with an account name and password. The account name and its password are created in the Windows Azure management Portal.

This implementation does not provide any built-in fault tolerance mechanism. It is possible, however, for the Client Application, do devise a fault-tolerant database system by using multiple CDB instances and picking different Local Storage Containers to work as fault-tolerance points. The Cloud Database Container is inherently fault-tolerant as a service provided by the Windows Azure Cloud. The development of a fault-tolerance mechanism would be adding another software layer to the existing API, and it is outside the scope of this work.

The CDB system currently does not provide a versioning system. Persisted files – main Index and Metadata - currently don't have a version number stored on them. For future work this issue will be addressed, so CDB systems of different versions will be able to coexist.