6 IMPLEMENTATION FOR MOBILE SYSTEMS

The API implementation for mobile devices is based upon Windows Phone 7.5 Operating System and .NET Framework 4.0.

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

The Cloud Storage Manager component is the Windows Azure Storage, and its access/manipulation is done via a Client API in the Microsoft.Samples.WindowsPhoneCloud.StorageClient library.

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

6.1. Windows Phone Execution Model and Data Persistence

A Windows Phone **Application** is similar to a Silverlight [26] Web Application in the sense that it is built upon **Pages**. In fact, the Windows Phone Programming model derives directly from the Silverlight Programming Model, originally conceived to run in Web browsers.

When a user starts an application from the installed applications list or from a tile on the Start button, the application Main Page is displayed. From the Main Page she can then navigate to other pages, which are other parts of the application. From the user's viewpoint, an application is comprised of one or more "screens". Each of these screens is modeled as a "page" on a Windows Phone application.

Differently, however, from the Silverlight Programming Model (which shall not be discussed here), is the Windows Phone Execution Model, which governs the lifecycle of applications running on a Windows Phone. In this model, an application can be, at any time, in the following **State**:

- Running: The Application was started and it's running normally.

- Deactivated: The application becomes deactivated when the user navigates forward away from the application, by pressing the Start button or by launching another application. ***Any application data should be saved to the persistent storage so it could be restored at later time***.
- Tombstoned: Immediately after the application is deactivated, the operating system will attempt to put the application into a tombstoned state. In this state, all of the application's threads are stopped, but the application remains intact in memory. If the application is reactivated from this state, the application does not need to reload any data into memory, because it has been preserved. In this state, application reactivation is faster, but tombstoning requires that extra memory be available.

A very important characteristic that governs this execution model is that when an application changes from Running state to a Deactivated/Tombstoned state, a limit of 10 seconds is set for the application to complete any finalization or data-saving task, through system-provided event handlers. ***If an application exceeds this limit, it will be immediately terminated.*** Therefore, operations that could last long, like reading or writing to the persistent storage, should be done throughout the lifetime of the application; in other words, relevant data should be persisted as soon as it changes. As a consequence, CRUD operations performed by the CDB in this implementation avoid batch/cache strategies; i.e., any time an Add, Update or Delete action performed by the a CDB instance needs to write to the Main Index File or to the Metadata file, it will write the data immediately to the Application Isolated Storage.

6.2.Windows Phone Thread Model – need for Asynchrony

6.2.1.UI Thread, Composition Thread and Thread Pool

Windows Phone 7 applications follow the Thread Model below:

- **UI thread** – Most important thread, it handles parsing and rendering of UI elements from XAML [27] elements; executes the UI elements

event handlers and callbacks associated to them. It is the most important thread in any application.

- **Composition thread** – Used to offload the UI thread helping with graphics and animations, working with the GPU.
- **Thread Pool** - Any application can also use a Thread Pool for multitasking or asynchronous operations, generally used to perform long-running background operations that do not block the UI thread. The Thread Pool consists of 2 threads per CPU on the mobile device.

The Composition thread, important on gaming applications, is not used on the CDB implementation. What is utterly important to it is the fact ***that the UI thread cannot be blocked on a long-running operation for more than 10 seconds. If it happens, the Operating System will terminate the application***.

The immediate consequence of this limitation is that all CDB CRUD operations have to be implemented based on an asynchronous pattern, as we will see in the next section, and they rely on the Thread Pool on this implementation model.

6.2.2.CDB Operations Implemented Asynchronously

6.2.2.1.Asynchrony on a Nutshell

Calling an asynchronous method means that the method, no matter how long it takes to execute, will return immediately the flow of execution to the caller thread after its call. The execution is spawn on another thread and the flow of execution on the caller method continues immediately on the following line. Let's consider, for instance, fictitious method MyMethod(), running on the UI thread, which contains an Asynchronous method DoSecondAsync():

```
DoFirst();
var myResult = DoSecondAsync();  // Asynchronous method;
DoThird(myResult);
```

In this case, DoSecondAsync() will be started on another thread and the flow of execution will resume immediately on MyMethod() to method DoThird(),

while DoThid() and DoSecondAsync() will be running concurrently. As an immediate consequence, the call to DoThird() using a variable computed by DoSecondAsync() is most likely a serious error, for it is not guaranteed that DoSecondAsync() will be even started when DoThird() is already finished – it is up to the Thread Pool scheduler to decide when to run it.

In order to be able to utilize the results of an asynchronous method as soon as it is finished, as well as to deal with possible exceptions thrown by asynchronous methods (which are not thrown in the caller thread and therefore need to be explicitly looked after), there are a few implementations for known patterns for Asynchronous Programming publicly available. This implementation uses the IAsyncResult [28] pattern, which shall not be discussed here, but here's its central idea:

- When you call a method that starts asynchronously and follows the IAsyncResult pattern, you pass along a callback function as a parameter;
- The method runs asynchronously on another thread and it either completes successfully or aborts throwing an exception;
- Once the method is finished, it will call the callback function and its *state* (any data relevant to the application) is made available for the callback function to use.

6.2.2.2.Asynchronous Methods on the Windows Phone 7 CDB API

Implementing CDB operations asynchronously brings two major consequences to the overall API development. The first one is that the implementation of Add/Delete methods to the Cloud Storage ends up being a ***daisy-chain of callback functions***. To see how it works, let's consider the CDB method AddBlobToCloud(). A simplified pseudocode for this method, as implemented in the **desktop** API, follows:

```
AddBlobToCloud(filename)
  bool success = false;
  If (AddFileToCloud(filename) == true)
```

```
        If (SetFileMetadata(filename) == true)
          If (AddFileToMainIndex(filename) == true)
             Return true
Return success;
```

This natural, comfortable-to-the-eye flow of code breaks down when we implement AddFileToCloud() and SetFileMetadata() as asynchronous methods, for the reasons already explained. The Windows Phone Implementation for AddFileToCloud() will become more or less what's in the following pseudocode:

```
AddBlobToCloud(string filename, bool success)
   AddFileToCloud(filename, callback_SetMetadata, success);
```

And that's it – nothing else. Then, another piece of asynchronous code, named callback_SetMetadata, is defined somewhere else in the source code file (simplified pseudoCode):

```
callback_SetMetadata (result, callback_AddFileToMainIndex)
// result packs any data needed that was produced in
//   AddFileToCloud() so you can use to set the Metadata
```

And finally another piece of code, callback_AddFileToMainIndex, will finally update the Main Index, only after the file is safely stored in the Cloud, as well as its metadata:

```
callback_AddFileToMainIndex (result)
// result packs any data needed that was produced by
// the previous execution of callback_SetMetadata()
```

This daisy-chain of callback functions, while cumbersome to program and to debug, has no impact for the Client Developer. Another consequence of this approach, however, is very important to understand, which is the unreliable return values for the asynchronous methods AddBlobToCloud(*fileName*) and DeleteBlobFromCloud(*fileName*), as well as an explicit fix for it:

Even though AddBlobToCloud() and DeleteBlobFromCloud() were originally implemented as methods which return a Boolean value to indicate successful or failed operation, their asynchronous implementation can return any value even before they are fully completed, for they involve not only dealing with files but also with metadata and the CDB Main Index. To mitigate such a situation, these method signatures were changed to accommodate a Boolean parameter that will be set only when the last callback on the daisy-chain is executed (see Table 5).

Desktop Implementation
public bool AddBlobToCloud(Blob blob, string fileFullPath)
public bool DeleteBlobFromCloud(string filename)
Windows Phone Implementation
public bool AddBlobToCloud(Blob blob, string fileFullPath, **bool success**)
public bool DeleteBlobFromCloud(string filename, **bool success**)

Table 5: Checking for successful operations in the Cloud

It is still left to the Client Application to check this variable whenever needed. ***Again, this programming model will not block the UI thread in any moment*** – it's up to the Client Application to conform to the Asynchronous programming paradigm.

6.3.Types, persistence and access methods

6.3.1. Main index and Main index file

Main Index description – Similar to the Desktop implementation, the main index is implemented using a .NET 4.0 Dictionary provided by the .NET Framework 4.0 System.Generics.Collections library, with the same 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 readable, text 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.

NOTE: This implementation differs from the Desktop implementation for the lack of a mechanism to recover the Main Index after a crash – there is no Operations Log. It was left as a task for a future version of the Windows Phone CDB.

The main Index File structure (see Table 6) is the same as the Desktop one, except that the file is persisted as clear, readable text, as opposed to a binary file found in the Desktop implementation:

Name convention	CDB_*DatabaseName*.IND		
File Type	Text file, records with 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 6: Main Index File Structure

6.3.2.Metadata and Metadata File

Similar to the desktop implementation, blob metadata is stored in a metadata file specific to its entity class; 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. One application that defines N blob-derived Entity classes will use N metadata files.

Different from the Desktop implementation, the metadata file is persisted as a readable text file:

Name convention	*DatabaseName_DataTypeName* .TXT			
File Type	Text, records with varied size			
File sample				
\WP7CDB\Container\Brahms_01.m4a	Author	J. Brahms	Title	Von ewige Liebe
\WP7CDB\Container\Brahms_05.m4a	Author	J. Brahms	Title	Die Mainacht
\WP7CDB\Container\Brahms_16.m4a	Author	J. Brahms	Title	Hungarian Dance 5

Table 7: Metadata file for the Windows Phone API

6.4.Database limitations

On a Mobile device, each application is set aside a private storage area referred as to its Isolated Storage. One application cannot use another's application Storage Area, which provides a level of security for data access not present in desktop computers. However, for Cloud Storage, a very basic mechanism provided by the Azure CloudStorage API is provided – the same used on the desktop implementation.

Similar to the Desktop implementation, this one does not provide any built-in fault tolerance mechanism.

Similar to the Desktop implementation, this one currently does not provide a versioning system.