Aneka Tutorial Series

Developing MapReduce.NET Applications

Chao Jin and Christian Vecchiola

Abstract

This tutorial describes the MapReduce programming model in the .NET environment and explains how to create distributed applications with it over Aneka. It illustrates some examples provided with the Aneka distribution. It provides a detailed guide on how to create a MapReduce application by using the Microsoft Visual Studio 2005 Development Environment. After having read this tutorial, the users will be able to develop their own MapReduce.NET application over Aneka.

Document Status

Creation Date:	11/28/08
Version:	0.1
Classification:	User
Authors:	Chao Jin, Christian Vecchiola
Last Revision Date:	09/17/09
Status:	Draft

1. Prerequisites

In order to fully understand this tutorial the user should be familiar with the general concepts of Grid and Cloud Computing, Object Oriented programming and generics, distributed systems, and a good understanding of the .NET framework 2.0 and C#.

The practical part of the tutorial requires a working installation of Aneka. It is also suggested to have Microsoft Visual Studio 2005 (any edition) with C# package installed[1] even if not strictly required.

2. Introduction

Aneka allows different kind of applications to be executed on the same

1 Any default installation of Visual Studio 2005 and Visual Studio 2005 Express comes with all the components required to complete this tutorial installed except of Aneka, which has to be downloaded and installed separately.

Grid/Cloud infrastructure. In order to support such flexibility it provides different abstractions through which it is possible to implement distributed applications. These abstractions map to different execution models. Currently Aneka supports three different execution models:

- *Task Execution Model*
- *Thread Execution Model*
- *MapReduce Execution Model*

Each execution model is composed by four different elements: the *WorkUnit*, the *Scheduler*, the *Executor*, and the *Manager*. The *WorkUnit* defines the granularity of the model; in other words, it defines the smallest computational unit that is directly handled by the Aneka infrastructure. Within Aneka, a collection of related work units define an application. The *Scheduler* is responsible for organizing the execution of work units composing the applications, dispatching them to different nodes, getting back the results, and providing them to the end user. The *Executor* is responsible for actually executing one or more work units, while the *Manager* is the client component which interacts with the Aneka system to start an application and collects the results. A view of the system is given in Figure 1.

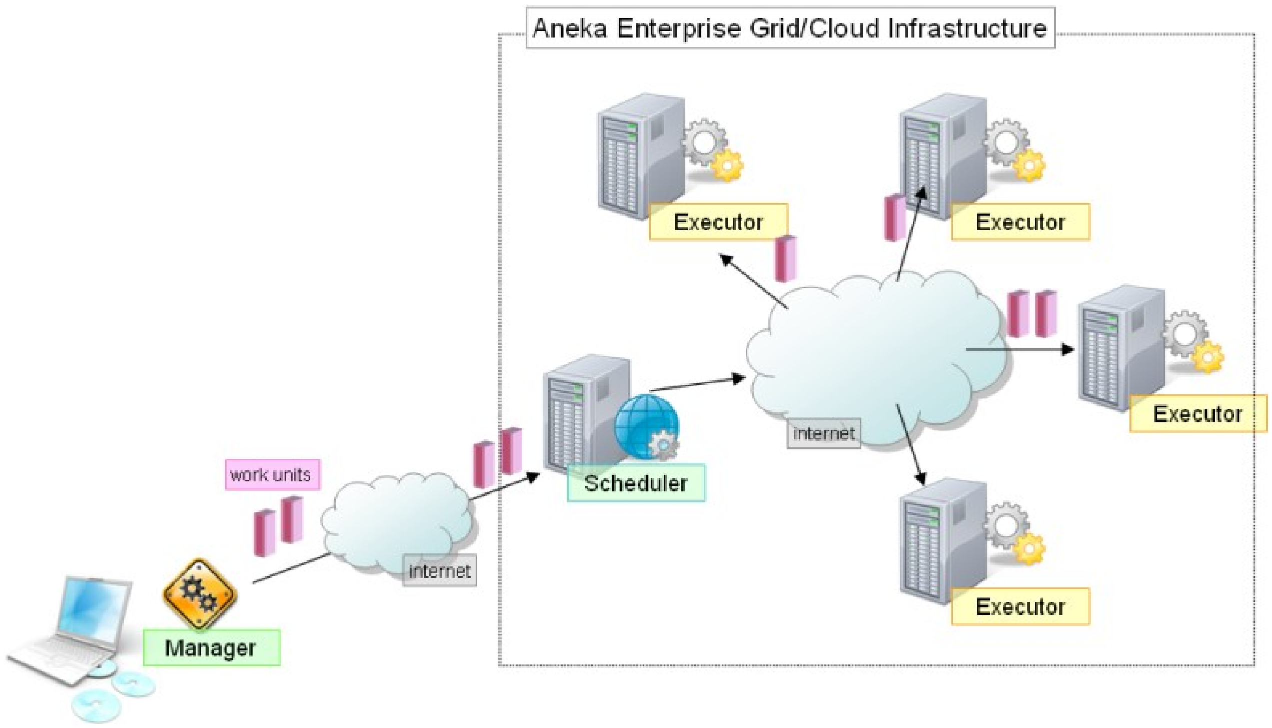


Figure 1: System Component View

In the case of MapReduce the user does not directly express the tasks that need to be computed but specifies two operations that are applied to the tasks generated automatically by the infrastructure. Users will not directly deal with a specific *WorkUnit*, but we will still have a *MapReduce Scheduler*, a *MapReduce Executor*, and a *MapReduce Manager*. In order to develop an application for Aneka, the user does not have to know all these components; Aneka handles a lot of the work by itself without the user's contribution. Only few things users are required to know:

- how to define *Map* and *Reduce* operations specific to the application that is being defined;
- how to create a *MApReduceApplication* and use it executing MapReduce.NET applications;
- how to control the *MapReduceApplication* and collect the results.

In the remainder of this tutorial will then concentrate on the *MapReduce Model*, but many of the concepts described can be applied to other execution models.

3. MapReduce Model

3.1 MapReduce Overview

MapReduce is triggered by *map* and *reduce* operations in functional languages, such as Lisp. This model abstracts computation problems through two functions: map and reduce. All problems formulated in this way can be parallelized automatically.

All data processed by *MapReduce* are in the form of key/value pairs. The execution happens in two phases. In the first phase, a map function is invoked once for each input key/value pair and it can generate output key/value pairs as intermediate results. In the second one, all the intermediate results are merged and grouped by keys. The reduce function is called once for each key with associated values and produces output values as final results.

3.2 Map and Reduce

A map function takes a key/value pair as input and produces a list of key/value pairs as output. The type of output key and value can be different from input key and value:

$$map::(key_1,value_1) => list(key_2,value_2)$$

A reduce function takes a key and associated value list as input and generates a list of new values as output:

$$reduce::list(key_2,value_2) => list(value_3)$$

3.3 MapReduce Execution

A *MapReduce* application is executed in a parallel manner through two phases. In the first phase, all map operations can be executed independently with each other. In the second phase, each reduce operation may depend on the outputs generated by any number of map operations. However, similar to map operations, all reduce operations can be executed independently.

From the perspective of dataflow, *MapReduce* execution consists of *m* independent map tasks and *r* independent reduce tasks, each of which may be dependent on *m* map tasks. Generally the intermediate results are partitioned into *r* pieces for *r* reduce tasks.

The *MapReduce* runtime system schedules map and reduce tasks to distributed resources. It manages many technical problems: parallelization, concurrency control, network communication, and fault tolerance. Furthermore, it performs several optimizations to decrease overhead involved in scheduling, network communication and intermediate grouping of results.

4. MapReduce.NET

MapReduce is one of most popular programming models designed for data centers. It supports convenient access to the large scale data for performing computations while hiding all low level details of physical environments. It been proved to be an effective programming model for developing data mining, machine learning and search applications in data centers. Especially, it can improve the productivity for those junior developers without required experiences of distributed/parallel development.

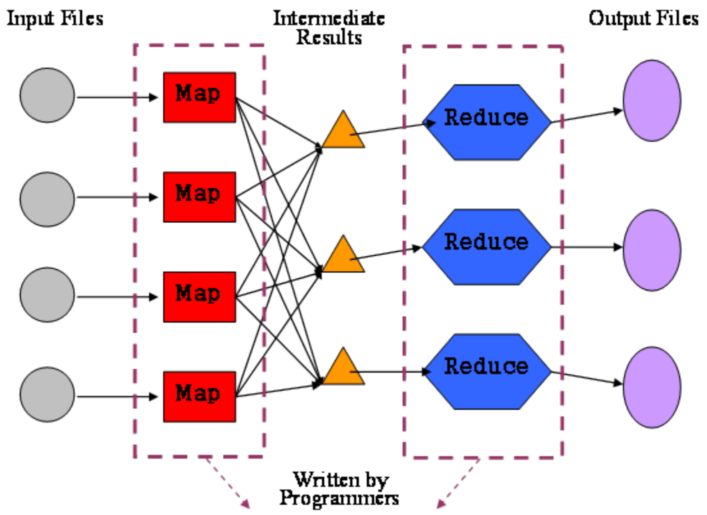


Figure 2: MapReduce Execution Model

MapReduce.NET is an implementation of *MapReduce* for data centers and resembles Google's MapReduce with special emphasis on the .NET and Windows platform. *MapReduce.NET* supports an object-oriented interface for programming *map* and *reduce* functions, and includes a set of storage APIs to wrap input key/value pairs into initial files and extract result key/value pairs from results files. Figure 2 illustrates the big picture of *MapReduce.NET* from the point view of users.

4.1 Architecture

The design of MapReduce.NET aims to reuse as many existing Windows services as possible. Figure 3 illustrates the architecture of MapReduce.NET. Our implementation is assisted by several distributed component services of Aneka.

MapReduce.NET is based on master-slave architecture. Its main components include: manager, scheduler, executor and storage.

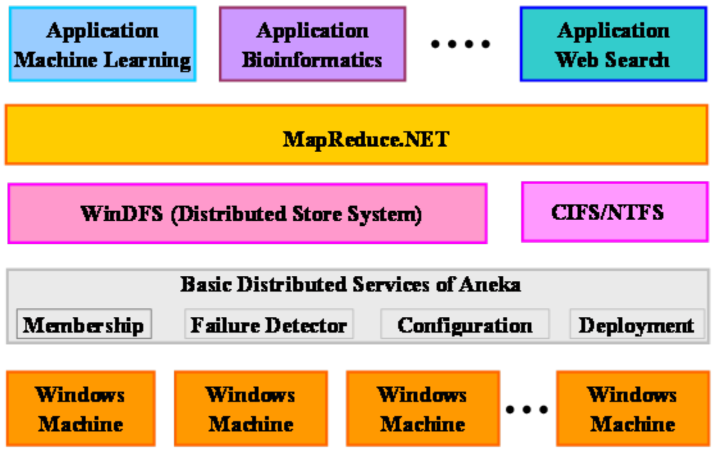


Figure 3: MapReduce Archtecture

- *Manager*: the manager works as an agent of MapReduce computation. It submits applications to the MapReduce scheduler and collects the final results after the execution completes successfully.
- *Scheduler*: after users submit MapReduce.NET applications to the scheduler, it maps sub tasks to available resources. During the execution, it monitors the progress of each task and takes corresponding task migration operation in case some nodes are much slower than others due to heterogeneity.
- *Executor*: each executor waits task execution commands from the scheduler. For a Map task, normally its input data locates locally. Otherwise, the executor needs to fetch input data from neighbors. For a Reduce task, the executor has to fetch all the input and merge them before execution. Furthermore, the executor monitors the progress of executing task and frequently reports the progress to the scheduler.
- *Storage*: the storage component of MapReduce.NET provides a distributed storage service over the .NET platform. It organizes the disk spaces on all the available resources as a virtual storage pool and provides an object based interface with a flat name space, which is used to manage data stored in it.

Figure 4 illustrates an example configuration of MapReduce.NET deployment with Aneka. Client processes simply contain the libraries required to connect to Aneka and to submit the execution of MapReduce tasks. A typical configuration will have installed the MapReduce scheduler on the scheduler node while Mapreduce executors will be deployed on executors node. Each of the executor nodes will access a shared storage among all the MapReduce executors by means of the Windows Shared File Service.

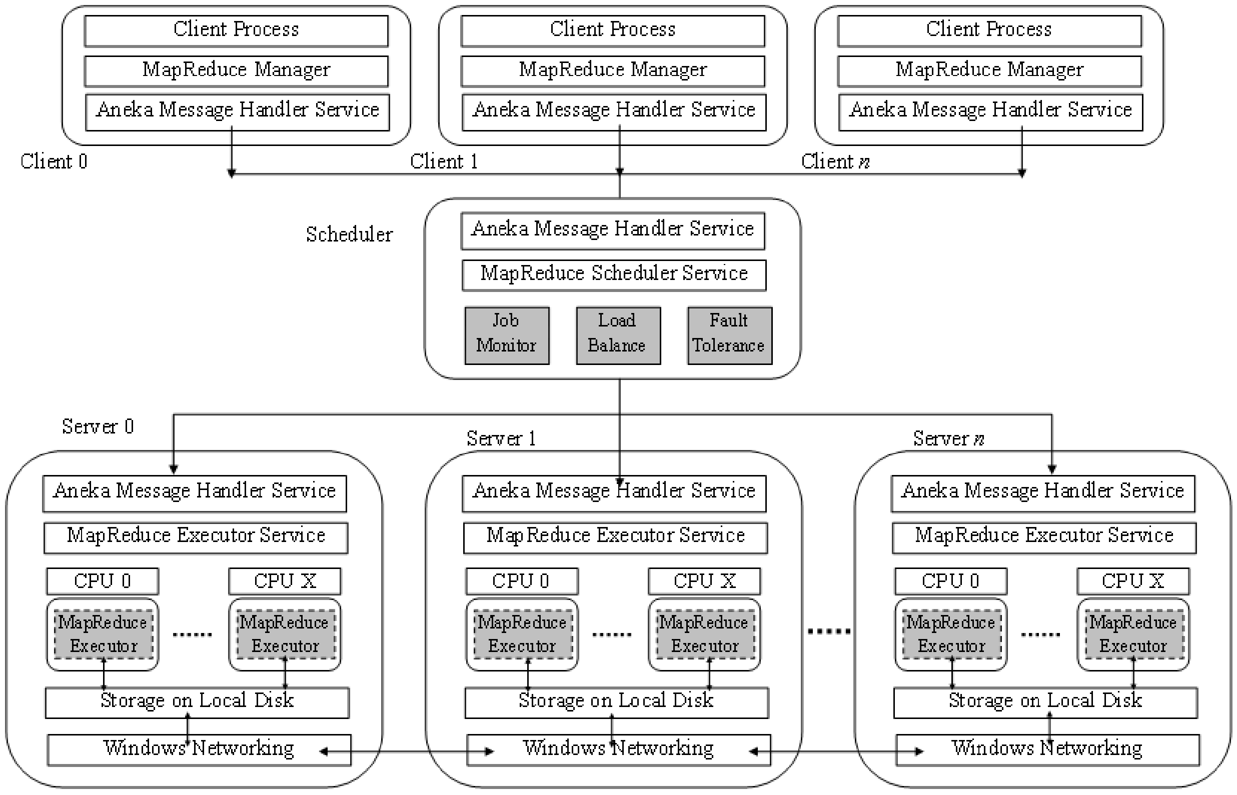


Figure 4: MapReduce System View

4.2 MapReduce.NET API

The implementation of MapReduce.NET exposes APIs similar as Google MapReduce. A MapReduce.NET based application is then composed by three main components:

- *Mapper*: this class represents the base class that users can inherit to specify their map function.
- *Reducer*: it represents the base class that uses can inherit to specify their reduce function.
- *MapReduceApplication*: represents the driver of the entire application. This class is configured with the Mapper and the Reducer specific classes defined by the user and is in charge of starting the execution of MapReduce.NET, controlling it and collecting the results.

Section 4.2.1 and 4.2.2 will respectively describe in detail the API for implementing the Map function and the Reduce functions. Section 4.2.3 will then present the API required to execute a MapReduce.NET application and how to configure them with the specific *Map* and *Reduce* functions.

4.2.1 Map API

```
namespace Aneka.MapReduce
{
      /// <summary>
      /// Interface MapInput. Represents a key/value pair.
      /// </summary>
      public interface MapInput<K,V> : MapInput
      {
         /// <summary>
         /// Gets the Key of the key/value pair.
         /// </summary>
         K Key { get; }
         /// <summary>
         /// Gets the value of the key/value pair.
         /// </summary>
         V Value { get; }
      }

      /// <summary>
      /// Delegate MapEmitDelegate. Defines the signature of the method that
      /// is called to submit intermediate results.
      /// </summary>
      /// <param name="key">key</param>
      /// <param name="value">value</param>
      public delegate void MapEmitDelegate(object key, object value);

      /// <summary>
      /// Class Mapper. Defines the base class that user can specialize to implement
      /// the map function.
      /// </summary>
      public abstract class Mapper<K,V> : MapperBase
      {
         /// <summary>
         /// Defines the Map function.
         /// </summary>
         /// <param name="input">input value</param>
         protected abstract void Map(MapInput<K,V> input);
     }
}
```

Listing 1 - Map Related Classes and Interfaces.

To define a map function, users have to inherit the *Mapper* class and then override the abstract *Map* function. The input argument is a *MapInput*, which contains one input key/value pair. The type of key and value are specified by generic parameters, *K* and *V* respectively.

The map function defined by users is invoked once for each input key/value pair. Normally, the map function produces another key/value pair as output. Moreover, each input key/value pair might generate a list of output key/value pairs. The generated intermediate results are collected by the *MapReduce.NET* runtime system by using *MapEmitDelegate*. Its input argument includes a key and a value for the intermediate key/value pair.

All of the collected intermediate key/value pairs are sorted by keys. Moreover, all values associated same key are grouped together. Each group of intermediate values are taken as an input for the reduce function, which normally performs an aggregate operation on the grouped values and produce a final result.

4.2.2 Reduce API

```
namespace Aneka.MapReduce
{
     /// <summary>
     /// Interface IReduceInputEnumerator. Defines an enumerator that iterates
     /// over the values of the same key.
     /// </summary>
     public interface IReduceInputEnumerator<V> : IEnumerator<V>,
                                                  IreduceInputEnumerator;
     /// <summary>
     /// Delegate ReduceEmitDelegate. Defines the signature of the method that
     /// is called to submit aggregate results.
     /// </summary>
     /// <param name="value">value</param>
     public delegate void ReduceEmitDelegate(object value);

     /// <summary>
     /// Class Reducer. Defines the base class that user can specialize to implement
     /// the reduce function. The reduce function defined by users will be invoked
     /// to perform an aggregation operation on all the values associated with same
     /// intermediate key, which are generated by the map function.
     /// </summary>
     public abstract class Reducer<K,V> : ReducerBase
     {
        /// <summary>
        /// Defines the Reduce function.
        /// </summary>
        /// <param name="input">input value</param>
        protected abstract void Reduce(IReduceInputEnumerator<V> input);
    }
}
```

Listing 2 - Reduce Related Classes and Interfaces.

To define a reduce function, users have to inherit the *Reducer* class and then override the abstract *Reduce* function. The input argument is an *IReduceInputEnumerator*, which is an enumerator iterating over each element of one group of values associated with the same key. The *IReduceInputEnumerator* interface implements the *IEnumerator* interface. The type of key and value are specified by generic parameters, *K* and *V* respectively.

Within the reduce function, users extract each element of input values and then perform an aggregation operation on all the input values. The aggregated final result is submitted to the *MapReduce.NET* runtime system by using ReduceEmitDelegate. Its input argument is an object. Therefore, it can accept any type of data as the final result.

All the final results are partitioned into multiple segments. The partition is achieved by performing a hash function on each key of the final result. The space of keys is partitioned into multiple pieces according to the specifications of users, and correspondingly values locate in the same piece of key space consists of one partition. The final results are collected by users after the MapReduce computation is completed successfully.

4.2.3 MapReduceApplication API

To start executing a *MapReduce* application, users have to configure it by using the *MapReduceApplication* class. Listing 3 shows the main methods and properties of the *MapReduceApplication* class.

```
namespace Aneka.MapReduce
{
     /// <summary>
     /// Class MapReduceApplication. Configures a MapReduce application.
     /// </summary>
     public class MapReduceApplication<M, R> :
                         ApplicationBase<MapReduceManager<M, R>>
        where M : MapReduce.Internal.MapperBase
        where R : MapReduce.Internal.ReducerBase
     {
         /// <summary>
         /// Gets, sets a the number of partitions of the final results.
         /// </summary>
         public int Partitions {  get { ... } set { .. } }
         /// <summary>
         /// Gets, sets a boolean value that indicates whether the application
         /// should download the final result files to the local machine or not.
         /// </summary>
         public bool FetchResults { get { ... } set { .. } }
         /// <summary>
         /// Gets, sets a boolean value that indicates whether the application
         /// should synchronize the execution of reducer task or not.
         /// </summary>
         public bool SynchReduce { get { ... } set { .. } }
         /// <summary>
         /// Gets, sets a boolean value that indicates whether to combine results
         /// from the map phase in order to reduce the number of intermediate
         /// values.
         /// </summary>
         public bool UseCombiner { get { ... } set { .. } }
         /// <summary>
         /// Gets, sets a boolean value that indicates whether the input for the
         /// map is already loaded into the storage or not.
         /// </summary>
         public bool IsInputReady { get { ... } set { .. } }
         /// <summary>
         /// Gets, sets a the number of times to re-execute a failed task.
         /// </summary>
         public int Attempts { get { ... } set { .. } }
         /// <summary>
         /// Gets, sets a the name of the log file for the application.
         /// </summary>
         public string LogFile { get { ... } set { .. } }

         /// <summary>
         /// Fires when the MapReduce application is finished.
         /// </summary>
         public event EventHandler<ApplicationEventArgs> ApplicationFinished

         //  <summary>
         /// Initializes a MapReduce application.
         /// </summary>
         /// <param name="configuration">Configuration information</param>
         public MapReduceApplication(Configuration configuration) :
                  base("MapReduceApplication", configuration) { ... }
         //  <summary>
         /// Initializes a MapReduce application.
         /// </summary>
         /// <param name="displayName">Application name</param>
         /// <param name="configuration">Configuration information</param>
         public MapReduceApplication(string displayName,
                                     Configuration configuration) :
                  base(displayName, configuration) { ... }


         // from the ApplicatioBase class...
         /// <summary>
         /// Starts the execution of the MapReduce application.
         /// </summary>
         public override void SubmitApplication() { ... }
         /// <summary>
         /// Starts the execution of the MapReduce application.
         /// </summary>
         /// <param name="args">Application finish callback</param>
         public override void InvokeAndWait(EventHandler<ApplicationEventArgs> args)
         { ... }

    }
}
```

Listing 3 - MapReduceApplication class public interface.

By looking at the content of Listing 3 we can identify three different elements that compose the interface of the *MapReduceApplication*. In the following we will explain their meaning and their function in the execution of *MapReduce.NET* applications.

Type Parameters

- *MapperBase* class: specifies the Mapper class defined by users.
- *ReducerBase* class: specifies the Reducer class defined by users.

Configuration Parameters

- *Partitions*: the intermediate results are partition into *r* pieces, which correspond to *r* reduce tasks. *ReducePartitionNumber* specifies *r*.
- *Attempts*: each map or reduce task may have errors during its execution. The errors may be caused by various types of reasons. Users can specify a number *n*, by using *Attempts*, to force the *MapReduce.NET* to re-execute its failed task *n* times for tolerating those faults that are not caused by the MapReduce application itself.
- *UseCombiner* and *SynchReduce* are parameters that control the behavior of the internal execution of MapReduce and are generally set to true.
- *LogFile* (optional): it is possible to store all the log messages generated by the application into a file for analyzing the execution of the application off line.

ApplicationBase Methods

- *SubmitApplication()* is called to submit MapReduce application and input files to the MapReduce scheduler and starts execution.
- *ApplicationFinished* is used to specify a function handler that is fired after the application is completed.

- *InvokeAndWait(EventHandler<ApplicationEventArgs> args)* this is a convenience method that is used to start the application and wait for its termination.

In order to create a *MapReduceApplication* it is necessary to specify the specific types of Mapper and Reduce that the application will use. This is done by specializing the template definition of the *MapReduceApplication* class. Moreovoer, in order to create an instance of this class it is necessary to pass an instance of the *Configuration* class, which contains all the customization settings for the application. *Configuration* is general class for all the programming models supported by Aneka and provides some features to support the definition of custom parameters.

```
namespace Aneka.Entity
{
    /// <summary>
    /// Class Configuration. Wraps the configuration parameters required
    /// to run distributed applications.
    /// </summary>
    [Serializable]
    public class Configuration
    {
       /// <summary>
       /// Gets, sets the user credentials to authenticate the client to Aneka.
       /// </summary>
       public virtual ICredential UserCredential { get { ... } set { .. } }
       /// <summary>
       /// If true, the submission of jobs to the grid is performed only once.
       /// </summary>
       public virtual bool SingleSubmission { get { ... } set { ... } }
       /// <summary>
       /// If true, uses the file transfer management system.
       /// </summary>
       public virtual bool UseFileTransfer { get { ... } set { ... } }
       /// <summary>
       /// Specifies the resubmission strategy to adopt when a task fails.
       /// </summary>
       public virtual ResubmitMode ResubmitMode { get { ... } set { ... } }
       /// <summary>
       /// Gets and sets the time polling interval used by the application to query
       /// the grid for job status.
       /// </summary>
       public virtual int PollingTime { get { ... } set { ... } }
       /// <summary>
       /// Gets, sets the Uri used to contact the Aneka scheduler service which is
       /// the gateway to Aneka grids.
       /// </summary>
       public virtual Uri SchedulerUri { get { ... } set { ... } }
```

```
        /// <summary>
        /// Gets or sets the path to the local directory that will be used
        /// to store the output files of the application.
        /// </summary>
        public virtual string Workspace { get { ... } set { ... } }
        /// <summary>
        /// If true all the output files for all the work units are stored
        /// in the same output directory instead of creating sub directory
        /// for each work unit.
        /// </summary>
        public virtual bool ShareOutputDirectory { get { ... } set { ... } }
        /// <summary>
        /// If true activates logging.
        /// </summary>
        public virtual bool LogMessages { get { ... } set { ... } }

        /// <summary>
        /// Creates an instance of the Configuration class.
        /// </summary>
        public Configuration() { ... }

        /// <summary>
        /// Loads the configuration from the default config file.
        /// </summary>
        /// <returns>Configuration class instance</returns>
        public static Configuration GetConfiguration() { ... }
        /// <summary>
        /// Loads the configuration from the given config file.
        /// </summary>
        /// <param name="confPath">path to the configuration file</param>
        /// <returns>Configuration class instance</returns>
        public static Configuration GetConfiguration(string confPath) { ... }

        /// <summary>
        /// Gets or sets the value of the given property.
        /// </summary>
        /// <param name="propertyName">name of the property to look for</param>
        /// <returns>Property value</returns>
        public string this[string propertyName] { get { ... } set { ... } }
        /// <summary>
        /// Gets or sets the value of the given property.
        /// </summary>
        /// <param name="propertyName">name of the property to look for</param>
        /// <param name="bStrict">boolean value indicating whether to raise
        /// exceptions if the property does not exist</param>
        /// <returns>Property value</returns>
        public string this[string propertyName, bool bStrict]
        { get { ... } set { ... } }
        /// <summary>
        /// Gets or sets the value of the given property.
        /// </summary>
        /// <param name="propertyName">name of the property to look for</param>
        /// <returns>Property value</returns>
        public string this[string propertyName] { get { ... } set { ... } }
        /// <summary>
        /// Gets the property group corresponding to the given name.
        /// </summary>
        /// <param name="groupName">name of the property group to look for</param>
        /// <returns>Property group corresponding to the given name, or
        /// null</returns>
        public PropertyGroup GetGroup(string groupName) { ... }
        /// <summary>
        /// Adds a property group corresponding to the given name to the
        /// configuration if not already present.
        /// </summary>
        /// <param name="groupName">name of the property group to look for</param>
        /// <returns>Property group corresponding to the given name</returns>
        public PropertyGroup AddGroup(string groupName) { ... }
        /// <summary>
        /// Adds a property group corresponding to the given name to the
        /// configuration if not already present.
        /// </summary>
        /// <param name="group">name of the property group to look for</param>
        /// <returns>Property group corresponding to the given name</returns>
        public PropertyGroup AddGroup(PropertyGroup group) { ... }
        /// <summary>
        /// Removes the group of properties corresponding to the given name from the
        /// configuration if present.
        /// </summary>
        /// <param name="groupName">name of the property group to look for</param>
        /// <returns>Property group corresponding to the given name if successfully
        /// removed, null otherwise</returns>
        public PropertyGroup RemoveGroup(string groupName) { ... }

        /// <summary>
        /// Checks whether the given instance is a configuration object and
        /// whether it contains the same information of the current instance.
        /// </summary>
        /// <param name="other">instance to compare  with</param>
        /// <returns>true if the given instance is of type Configuration
        /// contains the same information of the current instance.</returns>
        public override bool Equals(object other) { ... }

      }
}
```

Listing 1 - Configuration class public interface.

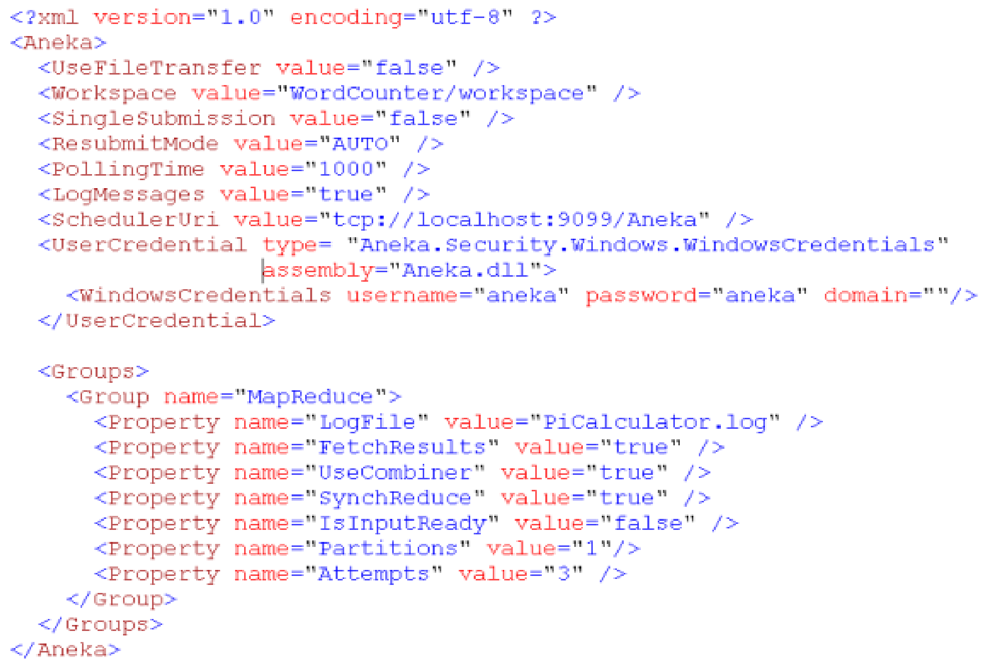
Listing 4 reports the public interface of the *Configuration* class. An instance of the Configuration can be created programmatically or by reading the application configuration file that comes with any .NET executable application. In case we provide the configuration parameters through the application configuration file it is possible to get the corresponding *Configuration* instance simply by calling the static method *Configuration.GetConfiguration()* or by using the overloaded version that allows us to specify the path of the configuration file. These methods expect to find an XML file whose structure is shown in Figure 5.

As it can be noticed the specific parameters required by the MapReduce model are contained in a specific tag <Group name="MapReduce"> ...</Group>. The properties contained in this tag are the same ones excosed by the *MapReduceApplication* class. The user can then customize the execution of the *MapReduce* model by simply editing the values of the properties contained in this file. The runtime will automatically, read this configuration values and link them to the properties of the *MapReduceApplication* class. It is possible to omit some of the properties contained in this tag or even omit the entire tag. In these cases the runtime will automatically update the configuration with the missing parameters by providing the default values.

For what concerns the *MapReduce* programming model the only other parameters that are of interest in the *Configuration* are: *SchedulerUri* and the *Workspace*. The first one identifies the uri of the Aneka scheduler while the second one points to the local directory that is used to store the output files. All the other parameters directly exposed by the *Configuration* class are of a general use for the other models but are not relevant for the execution of *MapReduce.NET*.

NOTE: The level of integration of MapReduce into the architecture of Aneka is still at an early stage. For this reason, some of the components that are implemented in the model do not comply with the general programming and design guidelines of the framework. For example MapReduce has a custom and optimized storage management system, which is separated from the storage service provided by the framework. Another feature that differs strongly from the other implementations is the scheduling service, which, at the moment, cannot be configured for supporting the persistence mode configured with the framework.

The configuration of the MapReduce.NET application is an example of the extensibility model of the *Configuration* class. It is possible to inetgrate user properties to the configuration by organizing them into groups and adding the groups to the configuration file or the *Configuration* instance. Once added to the file the runtime will automatically load them into the configuration object and expose them in the following format: *GroupName.PropertyName*



```
<?xml version="1.0" encoding="utf-8" ?>
<Aneka>
  <UseFileTransfer value="false" />
  <Workspace value="WordCounter/workspace" />
  <SingleSubmission value="false" />
  <ResubmitMode value="AUTO" />
  <PollingTime value="1000" />
  <LogMessages value="true" />
  <SchedulerUri value="tcp://localhost:9099/Aneka" />
  <UserCredential type= "Aneka.Security.Windows.WindowsCredentials"
                  assembly="Aneka.dll">
    <WindowsCredentials username="aneka" password="aneka" domain=""/>
  </UserCredential>

  <Groups>
    <Group name="MapReduce">
      <Property name="LogFile" value="PiCalculator.log" />
      <Property name="FetchResults" value="true" />
      <Property name="UseCombiner" value="true" />
      <Property name="SynchReduce" value="true" />
      <Property name="IsInputReady" value="false" />
      <Property name="Partitions" value="1"/>
      <Property name="Attempts" value="3" />
    </Group>
  </Groups>
</Aneka>
```

Figure 5: MapReduce Archtecture

By using this syntax it is possible to retrieve them by using the indexer method exposed by the Configuration class. For example if we want to retrieve the UseCombiner of the MapReduce group we could use the following statements:

```
Configuration conf = Configuration.GetConfiguration();
string useCombinerProperty = conf["MapReduce.UseCombiner"];
```

This is basically what the *MapReduceApplication* does with some additional casts and checks. Specific applications can create groups and store the parameters they need into the configuration file very easily. The *Configuration* class provides specific API for adding, removing, and retrieving *PropertyGroup* objects. Please refer to the documentation of the API for more details.

4.2.4 File API

MapReduce applications work intensively with files. The infrastructure stores and reads all the values that needs to execute Map and Reduce tasks from files. This is done in a completely transparent manner to the developer that does not have to care about file management. Hence, it is important to have a brief overview of the API that the model provides for file management.

Given the specific way in which data are manipulated the model there is only one type of file that is used by the infrastructure: sequence file. A sequence file contains a list of key/value pairs and it is represented by the

Aneka.MapReduce.DiskIO.SequenceFile class. If needed developers can read and write a sequence file by using the *SeqReader* and the *SeqWriter* classes whose public interface is listed in Listing 5.

```
namespace Aneka.MapReduce.DiskIO
{
     /// <summary>
     /// Class SeqReader. Provides an enumerator to read key/value pairs in a
     /// Sequence File.
     /// </summary>
     public class SeqReader
     {
        /// <summary>
        /// Creates a SeqReader instance for accessing the
        /// data contained in the file passed as parameter.
        /// </summary>
        /// <param name="file">path to the file to read</param>
        public SeqReader(string file) { ... }
        /// <summary>
        /// Set the type of key/value pair in the contained in the file.
        /// </summary>
        /// <param name="keyType">type of the key istances in the file</param>
        /// <param name="valueType">type of the value instances in the file</param>
        public void SetType(Type keyType, Type valueType) { ... }
        /// <summary>
        /// Closes the reader and releases the resources allocated
        /// for reading the sequence file.
        /// </summary>
        public void Close() { ... }
        /// <summary>
        /// Moves the reader to the next key/value pair.
        /// </summary>
        /// <returns>true if there are still key/value pairs, false otherwise
        /// </returns>
        public bool HaxNext() { ... }
        /// <summary>
        /// Return the current key.
        /// </summary>
        /// <returns>current value of the key.</returns>
        public Object NextKey() { ... }
        /// <summary>
        /// Return the current value.
        /// </summary>
        /// <returns>current value.</returns>
        public Object NextValue() { ... }
     }
     /// <summary>
     /// Class SeqWriter. Provides an interface to write key/value pairs to a
```

```
      /// SequenceFile. The only way to write a SequenceFile is to add the new
      /// key/value pair to the tail of the file.
      /// </summary>
      public class SeqWriter
      {
         /// <summary>
         /// Creates an instance of SeqWriter for writing to file.
         /// </summary>
         /// <param name="file">path to the file to write</param>
         public SeqWriter(string file) { ... }
         /// <summary>
         /// Closes the SeqWriter and releases all the resources allocated for
         /// writing to the sequence file.
         /// </summary>
         public void Close() { ... }
         /// <summary>
         /// Append the key/value pair to the taile of the SequenceFile.
         /// </summary>
         /// <param name="key">key value to add to the file</param>
         /// <param name="value">value to add to the file</param>
         public void Append(Object key, Object value) { ... }
      }

}
```

Listing 2 - SeqReader and SeqWriter public interfaces.

The *SeqReader* class opens a sequence file and supports an enumerator-like method to access the key/value pairs in the file. Users can check if there is still key/value pairs by *HasNext()*, and get key and value by *NextKey()* and *NextValue()* seperately. Before reading the sequence file, users have to specify the correct types of key/value pair by using *SetType(...)*. *Close()* is used to close the sequence file and its reader.

The *SeqWriter* class opens a sequence file and writes key/value pairs into the file. The only way to write content into a sequence file is by using *Append(key, value)*. Each key/value pair is appended to the tail of the sequence file. *Close()* is used to close the sequence file and its writer.

5. MapReduce Application Deployment

5.1 Overview

The deployment of *MapReduce* application is simplified by Aneka. In order to execute a *MapReduce.NET* application it is sufficient to create a *MapReduceApplication* class and specialize the template with the desired *Mapper* and *Reducer* that characterize the application. The specific behavior of the is then controlled by the *Configuration* object that can be loaded from a file (see section 4.2.3 for details).

Once the MapReduceApplication has ben set up it is possible to execute the application by invoking the *MapReduceApplication.SubmitApplication()* method. This method submits all input files that the application requires to be execute and then starts the execution of the application.

During execution, the status of the application is constantly monitored until it completes successfully or fails. If the application completes successfully, *MapReduceApplication* collects back the result files and put them into a local directory if the *FetchResults* property is set to true. The location into which the files are placed is defined by the Workspace property of the configuration. In case that the application cannot complete due to any failures, the reason of failure is sent back to users. The result of execution is implemented by the *ApplicationEventArgs* class, which contains the name list of result files.

5.2 Configuring the MapReduce.NET Infrastructure

The MapReduce.NET inrastructure is composed – as any other programming model supported by Aneka – of a scheduler service and an executor service. The only difference is the use of the storage. MapReduce.NET as a specific and optimized storage that is only used to store the files required by MapReduce applications.

In order to configure the components of the model it is possible to use the configuration wizard integrated into the Manager Console or the Aneka MSI installer (please see the Aneka Installation Guide for more details). We will briefly review the configuration settings for these components:

<u>MapReduceScheduler</u>

- *Username*: user name of the account who has the rights to execute MapReduce service.
- *Password*: password for the account who has the rights to execute MapReduce service.

<u>MapReduceExecutor</u>

- *Username*: user name of the account who has the rights to execute MapReduce service.
- *Password*: password for the account who has the rights to execute MapReduce service.
- *StorageDirectory*: local directory that is used to stage files by the *MapReduceExecutor*.

For what concerns the storage component there is no need of any specific configuration.

It is now possible to execute MapReduce application on Aneka and try the examples introduced in the next section.

6. Examples

In order to get the feeling of how to program *MapReduce.NET* applications two examples are presented: *WordCounter* and *PiEstimator*.

6.1 WordCounter

WordCounter is a benchmark example for the *MapReduce* programming model. It counts the number of each word which appears in a large number of documents. Normally, the disk space required to store all the documents cannot be reached by one machine. Therefore, these documents distribute over a collection of machines.

The *WordCounter* application uses the *Map* and *Reduce* operations in order to respectively count the occurrences of one word into a document and to sum all the occurrences of the same words computed from different files. In order to implement the application two C# projects have been developed:

- *WordCounter.Library*: is a class library containing the definition of the mapper and the reducer classes required by the WordCounter application. The definition of these classes is contained in the WordCounter.cs file and displayed in Listing 6.
- *WordCounter*: this is a simple console program that contains the code required to initialize the MapReduceApplication with the previously defined mapper and reducer classes, load the configuration of the application, and execute it on Aneka. The listing of the driver program is displayed in Listing 7.

```
using System;
using Aneka.MapReduce;

namespace Aneka.Examples.MapReduce.WordCounter
{
     /// <summary>
     /// Class WordCounterMapper. Mapper implementation for the WordCounter
     /// application. The Map method reads the input and splits it into
     /// words. For each of the words found the word is emitted to the
     /// output file with a value of 1.
     /// </summary>
     public class WordCounterMapper : Mapper<string, string>
     {
        /// <summary>
        /// Checks whether the given character is a space or not.
        /// </summary>
        /// <param name="letter">test character</param>
        /// <returns>true if the character is a spece, false otherwise</returns>
        static bool IsSpace(char letter)
        {
            return !char.IsLetter(letter);
        }
        /// <summary>
        /// Reads the input and splits into words. For each of the words found
        /// emits the word as a key with a vaue of 1.
        /// </summary>
        /// <param name="input">map input</param>
        public override void Map(MapInput<string, string> input)
        {
            string value = input.Value;

            string[] words = value.Split(
                         " \t\n\r\f\"\'|!-=()[]<>:{}.#".ToCharArray(),
                         StringSplitOptions.RemoveEmptyEntries);

            foreach(string word in words)
            {
                this.Emit(word, 1);
            }
        }
    }
    /// <summary>
    /// Class WordCounterReducer. Reducer implementation for the WordCounter
    /// application. The Reduce method iterates all over values of the enumerator
    /// and sums the values before emitting the sum to the output file.
    /// </summary>
    public class WordCounterMapper : Reducer<string, int>
    {
        /// <summary>
        /// Iterates all over the values of the enumerator and sums up
        /// all the values before emitting the sum to the output file.
        /// </summary>
        /// <param name="input">map input</param>
        public override void Reduce(IReduceInputEnumerator<int> input)
        {
            int account = 0;

            while(reduceInput.MoveNext())
            {
                int value = reduceInput.Current;

                account += value;
            }
            this.Emit(account);
        }
    }
```

```
}
```

Listing 3 - WordCounterMapper and WordCounterReducer implementation.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

using Aneka.MapReduce;
using Aneka.Entity;
using Aneka.MapReduce.DiskIO;
using System.Diagnostics;

namespace Aneka.Examples.MapReduce.WordCounter
{
     /// <summary>
     /// Class Program. Main application driver of the WordCounter application.
     /// </summary>
     class Program
     {
        /// <summary>
        /// Reference to the configuration object.
        /// </summary>
        static Configuration configuration = null;
        /// <summary>
        /// Location of the configuration file.
        /// </summary>
        static string configurationFileLocation = "conf.xml";

        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name="args">program arguments</param>
        static void Main(string[] args)
        {
            try
            {
                // process the arguments
                ProcessArgs(args);

                Console.WriteLine("Use " + configurationFileLocation + ", to " +
                                 "specify different configuration file, please " +
                                 "use -c [config-file] option!");

                // get the configuration
                configuration =
                    Configuration.GetConfiguration(configurationFileLocation);

                // configure MapReduceApplication
                MapReduceApplication<WordCountMapper,
                                     WordCountReducer> application =
                    new MapReduceApplication<WordCountMapper,
                                             WordCountReducer>("WordCounter",
                                                              configuration);

                // invoke and wait for result
                application.InvokeAndWait(
                   new EventHandler<ApplicationEventArgs>(OnApplicationFinished));
            }
            catch(Exception ex)
            {
                Usage();
            }
        }
        /// <summary>
        /// Hooks the ApplicationFinished events and process the results
        /// if the application has been successful.
        /// </summary>
        /// <param name="sender">event source</param>
        /// <param name="e">event information</param>
        static void OnApplicationFinished(object sender, ApplicationEventArgs e)
        {
            if (e.Exception != null)
            {
                Console.WriteLine(e.Exception.Message);
            }
            else
            {
                ParseResult();
            }
            Console.WriteLine("Press enter to finish!");
            Console.ReadLine();
        }
        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name="args">program arguments</param>
        static void ProcessArgs(string[] args)
        {
            for(int i = 0; i < args.Length; i++)
            {
                switch(args[i])
                {
                    case "-c":
```

```
                    i++;
                    configurationFileLocation = args[i];
                    break;
                default:
                    break;
            }
        }
    }
    /// <summary>
    /// Displays a simple informative message explaining the usage of the
    /// application.
    /// </summary>
    static void Usage()
    {
        Console.WriteLine("WordCounter.exe -c configuration_file");
    }
    /// <summary>
    /// Parses the results obtained from the MapReduceApplication
    /// and dumps the them into a single file where for each key
    /// is put along with a value. Then starts the Notepad application
    /// to show the content of this file.
    /// </summary>
    static void ParseResult()
    {
        DirectoryInfo sources = null;

        try
        {
            sources = new DirectoryInfo(configuration.Workspace);
        }
        catch(Exception ex)
        {
            Console.WriteLine("Reason {0}", ex.ToString());

            return;
        }

        FileStream resultFile = new FileStream("WordResult.txt",
                                               FileMode.Create,
                                               FileAccess.Write);
        StreamWriter resultWriter = new StreamWriter(resultFile);

        FileInfo[] resultList = sources.GetFiles();
        foreach(FileInfo result in resultList)
        {
            SeqReader seqReader = null;

            try
            {
                    seqReader = new SeqReader(result.FullName);
                }
                catch(SequenceFileException ex)
                {
                    Console.WriteLine(ex.ToString());
                    return;
                }

                seqReader.SetType(typeof(string), typeof(int));

                while(seqReader.HaxNext())
                {
                    Object key = seqReader.NextKey();
                    Object value = seqReader.NextValue();

                    resultWriter.WriteLine("{0}\t{1}", key, value);
                }

                seqReader.Close();
            }

            resultWriter.Close();
            resultFile.Close();

            Console.WriteLine("Please open WordResult.txt to see the result!");

            StartNotePad("WordCounter.txt");
        }
        /// <summary>
        /// Starts the Notepad application and instructs it to open the
        /// the file pointed by the parameter name.
        /// </summary>
        /// <param name="name">file name to open</param>
        public static void StartNotePad(string name)
        {
            Process notepad = Process.Start("notepad", name);
        }
    }
}
```

Listing 4 - WordCounter driver application.

The *WordCounter* application is configured with the file *conf.xml* that can be found its directory. Except for the general parameters of the *Configuration* class and the properties set for the *MapReduce* runtime there are no other parameters requested.

6.2 PiCalculator

PiCalculator is an application that uses the montecarlo simulation to estimate the value of π. The estimation of π can be structured as a *MapReduce* application by adopting the following strategy: the *Map* operation selects a collection of random values (between 0 and 1) and checks whether the value locates inside the circle of radius equal to 1 if this happens it collects the value; the *Reduce* operation all the values that are accessed via the input are summed together. Once the distributed execution the client component of the application estimates the value of π.

In order to implement the application two C# projects have been developed:

- *PiCalculator.Library*: is a class library containing the definition of the mapper and the reducer classes required by the *PiCalculator* application. The definition of these classes is contained in the PiCalculator.cs file and displayed in Listing 8.
- *PiCalculator*: this is a simple console program that contains the code required to initialize the *MapReduceApplication* with the previously defined mapper and reducer classes, load the configuration of the application, and execute it on Aneka. The listing of the driver program is displayed in Listing 9.

```
using System;
using Aneka.MapReduce;

namespace Aneka.Examples.MapReduce.PiCalculator
{
     /// <summary>
     /// Class PiMapper. Mapper implementation for the PiCalculator
     /// application. The Map method generates the random number between
     /// 0.0 and 1.0. If the generated values locates inside the circle they
     /// are emitted to the output file.
     /// </summary>
     public class PiMapper : Mapper<long, long>
     {
        /// <summary>
        /// Random number generator.
        /// </summary>
        Random random = new Random();
        /// <summary>
        /// Number of elements inside the circle.
        /// </summary>
        long numInside = 0;
        /// <summary>
        /// Number of elements outside the circle.
        /// </summary>
        long numOutside = 0;

        /// <summary>
        /// Reads the input and splits into words. For each of the words found
        /// emits the word as a key with a vaue of 1.
        /// </summary>
        /// <param name="input">map input</param>
        public override void Map(MapInput<string, string> input)
        {
            long nSample = input.Key;

            for(long idx = 0; idx < nSample; idx++)
            {
                double x = random.NextDouble();
                double y = random.NextDouble();

                double d = (x - 0.5) * (x - 0.5) + (y - 0.5) * (y - 0.5);

                if (d > 0.25)
                {
                    numOutside++;
                }
                else
                {
                    numInside++;
                }
            }

            this.Emit((long)0, numInside);
        }
    }
    /// <summary>
    /// Class PiReducer. Reducer implementation for the WordCounter
    /// application. The Reduce method iterates all over values of the enumerator
    /// and sums the values before emitting the sum to the output file.
    /// </summary>
    public class PiReducer : Reducer<string, int>
    {
        /// <summary>
        /// Iterates all over the values of the enumerator and sums up
        /// all the values before emitting the sum to the output file.
        /// </summary>
        /// <param name="input">map input</param>
        public override void Reduce(IReduceInputEnumerator<long> input)
        {
            long numInside = 0;

            while(reduceInput.MoveNext())
            {
                numInside += reduceInput.Current;
            }
            this.Emit(numInside);
        }
    }
}
```

Listing 5 - PiMapper and PiReducer public interfaces.

```
using System;
using System.Collections.Generic;
using System.Trheading;
using System.IO;

using Aneka.MapReduce;
using Aneka.Entity;
using Aneka.MapReduce.DiskIO;


namespace Aneka.Examples.MapReduce.PiCalculator
{
     /// <summary>
     /// Class Program. Main application driver of the PiCalculator application.
     /// </summary>
     class Program
     {
        /// <summary>
        /// Reference to the configuration object.
        /// </summary>
        static Configuration configuration = null;
        /// <summary>
        /// Location of the configuration file.
        /// </summary>
        static string configurationFileLocation = "conf.xml";
        /// <summary>
        /// Number of mapper to use.
        /// </summary>
        static long numberOfMaps = 1;
        /// <summary>
        /// Number of samples to take.
        /// </summary>
        static long numberOfSamples = 1;

        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name="args">program arguments</param>
        static void Main(string[] args)
        {
            if (args.Length != 6)
            {
                Usage();
            }
            else
            {
                try
                {
                    ProcessArgs(args);

                    if (numberOfSamples <= 0)
                    {
                        Console.WriteLine("Please specify Map task number!");
                        Usage();
                    }
                    else if (numberOfMaps <= 0)
                    {
                        Console.WriteLine("Please specify sample number!");
                        Usage();
                    }
                    else
                    {
                        // create an aneka configuration
                        Configuration configuration =
                                      Configuration.GetConfiguration(args[1]);

                        string rootedPath =
                            Path.IsPathRooted(configuration.Workspace) ?
                                      configuration.Workspace :
                                      Path.GetFullPath(configuration.Workspace);

                        for(int idx = 0; idx < numberOfMaps; idx++)
                        {
                            string input = Path.Combine(rootedPath, "input-" + idx);
                            SeqWriter writer = new SeqWriter(input);
                            writer.Append(numberOfSamples, (long)0);
                            writer.Close();
                        }

                        // start PiCalculator MapReduce
                        MapReduceApplication<PiMapper, PiReducer> application =
                               new MapReduceApplication<PiMapper,
                                                        PiReducer>(configuration);

                        Console.WriteLine("Application is running");
                        application.InvokeAndWait(
                               new EventHandler<ApplicationEventArgs>
```

```
                                                   (OnApplicationFinished));
                    }
                }
                catch(Exception ex)
                {
                    Usage();
                }
            }
        }
        /// <summary>
        /// Hooks the ApplicationFinished events and process the results
        /// if the application has been successful.
        /// </summary>
        /// <param name="sender">event source</param>
        /// <param name="e">event information</param>
        static void OnApplicationFinished(object sender, ApplicationEventArgs e)
        {
            if (e.Exception != null)
            {
                Console.WriteLine(e.Exception.Message);
            }
            else
            {
                IList<string> results = (IList<string>)e.Data;
                foreach(string result in results)
                {
                    SeqReader reader = new SeqReader(result);
                    reader.SetType(typeof(long), typeof(long));

                    long numInside = -1;
                    if (reader.HaxNext())
                    {
                        numInside = (long) reader.NextValue();
                    }
                    reader.Close();

                    if (numInside > 0)
                    {
                        double estimate = (double)(numInside * 4.0) /
                                                  (numberOfMaps * numberOfSamples);

                        Console.WriteLine("Pi: {0}", estimate);
                    }
                }
            }
            Console.WriteLine("Press enter to finish!");
            Console.ReadLine();
        }
        /// <summary>
        /// Processes the arguments given to the application and according
        /// to the parameters read runs the application or shows the help.
        /// </summary>
        /// <param name="args">program arguments</param>
        static void ProcessArgs(string[] args)
        {
            for(int i = 0; i < args.Length; i++)
            {
                switch(args[i])
                {
                    case "-c":
                        i++;
                        configurationFileLocation = args[i];
                        break;
                    case "-n":
                        i++;
                        numberOfMaps = int.Parse(args[i]);
                        break;
                    case "-s":
                        i++;
                        numberOfSamples = long.Parse(args[i]);
                        break;
                    default:
                        break;
                }
            }
        }
        /// <summary>
        /// Displays a simple informative message explaining the usage of the
        /// application.
        /// </summary>
        static void Usage()
        {
            Console.WriteLine("PiEstimator.exe -c configuration_file " +
                                          "-n map# -s sample#");
        }
    }
}
```

Listing 6 - PiCalculator driver application.

The *PiCalculator* application is configured with the file conf.xml that can be found in the directory application. The *PiCalculator* requires two more parameters that are passed via the command line. A possible improvement of the application is moving these two parameters in a *PropertyGroup* named *PiCalculator* inside the configuration file and changing the code of the *Main* method in order to read the values from the configuration. This task is left to the reader.

7. Conclusions

In this tutorial we have introduced the *MapReduce Model* implemented in Aneka for running a set of *MapReduce* applications. Within the *MapReduce Model* a distributed application is expressed in terms of a two operations: *Map* and *Reduce*. The first operation transform a list of key/value pairs into another list of key/value pairs; the second operation reduces a pair composed by a key and a list of values into another lsit of values. The combination of Map and Reduce operations allows to process and elaborate large amount of data with a very strightforward approach. The MapReduce.NET is an implementaion of MapReduce for Aneka and the .NET framework and mades this abstraction available as a programming model for Aneka.

In order to create a *MapReduce* application it is necessary to create an instance of the *MapReduceApplication* properly configured with the mapper and reducer classes that define the operation of the application. This class constitute the client view of the *MapReduce* application and interacts with Aneka for executing the sequence of map and reduce operations by handling automatically scalability and fault tolerance.

This tutorial has covered the following arguments:

- General notions about the *MapReduce Model*.
- How to define a class that specialize the *Mapper<K,V>* class for defining the Map operation.
- How to define a class that specialize the Reducer*<K,V>* class for defining the Map operation.
- How to create a *MapReduceApplication* instance and configure it with the specific implementation of *MapperBase* and *ReducerBase*.
- How to tune the execution of *MapReduce* applications and starts their execution.
- How to customize the *Configuration* class with the custom values and how to manage the group of custom properties.
- How to collect the results from a *MapReduce* application.
- How to design and implement simple application such as *WordCounter* and *PiCalculator* by following the *MapReduce* model.

This tutorial does not fully cover what can be done with the MapReduce Model. For a more detailed information about this and other aspects the user can have a look at the APIs documentation.