Aneka Tutorial Series

Developing Task Model Applications

Christian Vecchiola and Xingchen Chu

Abstract

This tutorial describes the Aneka Task Execution Model and explains how to create distributed applications based on it. It illustrates some examples provided with the Aneka distribution which are built on top of the Task Model. It provides a detailed step by step guide for users on how to create a simple application that submit multiple tasks to Aneka and collect the results. After having read this tutorial the users will be able to develop their own application on top the Aneka Task Model.

Document Status

Creation Date:	05/01/08
Version:	0.1
Classification:	User
Authors:	Christian Vecchiola, Xingchen Chu
Contributors:	Alexandre di Costanzo Marcos Dias de Assunçao
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.

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.

2. Introduction

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

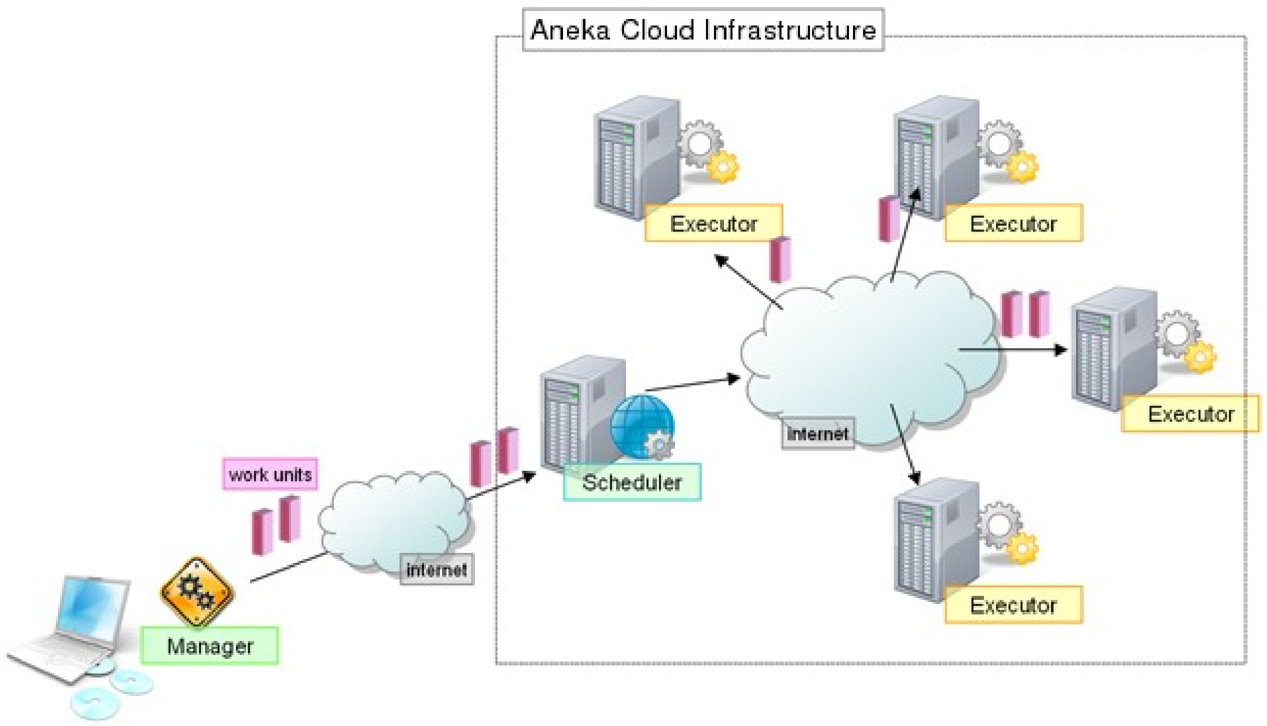


Figure 1. System Components View.

Hence, for the Task Model there will be a specific *WorkUnit* called *AnekaTask*, a *Task Scheduler*, a *Task Executor*, and a *Task 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 contribution. Only few things users are required to know:

- how to define *AnekaTask* instances specific to the application that is being defined;
- how to create a *AnekaApplication* and use it for task submission;
- how to control the *AnekaApplication* and collect the results.

This holds not only for the Task Model but for all execution models supported by the Aneka.

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

3. Task Model

3.1 What are Tasks?

The Task Model defines an application as a collection of *tasks*. Tasks are *independent work units* that can be executed in any order by the Scheduler. Within the Task Model a task comprises all the components required for its execution on a grid node.

The Task Model is the right solution to use when the distributed application consists of a collection of independent jobs that are executed on a grid and whose results are collected and composed together by the end user. In this scenario the user creates a set of tasks, submits them to Aneka, and waits for the results.

More complex scenarios are characterized by the dynamic creation of tasks while the application is executing, based on the previously executed tasks. Even in this case, from the Aneka point of view the tasks submitted for execution are independent and it is responsibility of the user to ensure the proper execution order and sequencing.

Tasks are sufficiently general abstractions that are useful to model the jobs of embarrassingly parallel problems. They provide a simple operation Execute through which the user can specify the computation that can be carried out by the task.

3.2 Working with Tasks

During the execution of the task there are two components that are involved: the *AnekaTask* class and the *ITask* interface.

The *AnekaTask* class represents the work unit in the Task Model. The user creates *AnekaTask* instances, configures them, and submit them to the grid by means of the *AnekaApplication* class. Aneka deals with AnekaTask instances which are then responsible for executing the job they contain.

```
namespace Aneka.Tasks
{
    /// <summary>
    /// Class AnekaTask. Represents the basic unit of work
    /// in the Task Model. It wraps a generic task to execute.
    /// </summary>
    public class AnekaTask : WorkUnit
    {
       ....
       /// <summary>
       /// Gets the User Task to execute.
       /// </summary>
       public ITask UserTask { get { ... } internal set { ... } }
       /// <summary>
       /// Gets a value indicating whether the current grid task
       /// is running. A grid WorkUnit is considered running if it
       /// is not Unstarted | Stopped | Completed | Rejected | Aborted.
       /// </summary>
       public bool IsRunning { get { ... } }
       ....
       /// <summary>
       /// Creates an instance of the AnekaTask.
       /// </summary>
       /// <param name="task">user task to execute.</param>
       public AnekaTask(ITask task)
       { ... }


    }
}
```

Listing 1 - AnekaTask class.

Listing 1 presents the public interface of the *AnekaTask* class. This is a framework class that is used to wrap the specific user task that will execute once scheduled.

In order to create a AnekaTask instance it is necessary to pass to the constructor a non null *ITask* instance. This instance represents the task that will be executed and it is exposed through the *UserTask* property. Listing 2 displays the ITask interface. This interface is very minimal and exposes only one method which is the *Execute* method. This method is called when the *AnekaTask* instance is executed on a grid node. Inside this method concrete classes have to define the code to perform what is needed for the specific distributed application that is being implemented.

```
namespace Aneka.Tasks
{
    /// <summary>
    /// Interface ITask. Defines a contract for task
    /// implementation. It provides an entry point
    /// for specific task execution.
    /// </summary>
    /// <remarks>
    /// Concrete classes must be marked Serializable
    /// or implement the ISerializable interface.
    /// </remarks>
    public interface ITask
    {
            /// <summary>
            /// Executes the task.
            /// </summary>
            public void Execute();
     }
}
```

Listing 2 - ITask interface.

Concrete classes implementing this interface must be marked Serializable or implement the *ISerializable* interface because *AnekaTask* instances need to be serialized in order to be transmitted over the network.

As shown above the creation of the work units of the Task Model is really simply and intuitive. The following steps summarize what is needed to create application specific grid tasks:

1. define a class *MyTask* which implements *ITask* and provide the specific code for the Execute method;
2. create as many *MyTask* instances as needed by your application;
3. wrap each *MyTask* instance into a *AnekaTask* instance by passing it to the *AnekaTask* constructor;
4. submit the *AnekaTask* instances to Aneka by means of the *AnekaApplication* class.

Listing 3 provides a template code implementing the steps described. In the example the *MyTask* class simply evaluates the value of a complex function for the given double value X, and exposes the result by means of the Value property. The example proposed in the listing uses the Normal distribution as test function. Obviously, this is a really trivial example meant only to show how to implement a task.

```
// File: MyTaskDemo.cs
using Aneka.Entity;
using Aneka.Tasks;

namespace Aneka.Examples.TaskDemo
{
    // Step 1: definition of the MyTask class.

    /// <summary>
    /// Class MyTask. Simple task function wrapping
    /// the Gaussian normal distribution. It computes
    /// the value of a given point.
    /// </summary>
    [Serializable]
    public class MyTask : ITask
    {
       /// <summary>
       /// value where to calculate the
       /// Gaussian normal distribution.
       /// </summary>
       private double x;
       /// <summary>
       /// Gets, sets the value where to calculate
       /// the Gaussian normal distribution.
       /// </summary>
       public double X  { get { return this.x; } set { this.x = value; } }
       /// <summary>
       /// value where to calculate the
       /// Gaussian normal distribution.
       /// </summary>
       private double result;
       /// <summary>
       /// Gets, sets the value where to calculate
       /// the Gaussian normal distribution.
       /// </summary>
       public double Result
       { get { return this.result; } set { this.result = value; } }
       /// <summary>
       /// Creates an instance of MyTask.
       /// </summary>
       public MyTask() {}
       #region ITask Members
       /// <summary>
       /// Evaluate the Gaussian normal distribution
       /// for the given value of x.
       /// </summary>
       public void Execute()
       {
          this.result = (1 / (Math.Sqrt(2* Math.PI))) *
                         Math.Exp(- (this.x * this.x) / 2);
       }
       #endregion
    }

    // Step 2-4: Task creation and submission

    /// <summary>
    /// Class Program. Simple Driver application
    /// that shows how to create tasks and submit
    /// them to the grid.
    /// </summary>
    /// <remarks>
    /// This class is a partial implementation and does
    /// not provide the complete steps for running the
    /// application on Aneka.
    /// </remarks>
    class MyTaskDemo
    {
       ....
       /// <summary>
       /// Program entry point.
       /// </summary>
       /// <param name="args">program arguments</param>
       public static void Main(string[] args)
       {
           AnekaApplication<AnekaTask,TaskManager> app = Setup(args);
           // create task instances and wrap them
           // into AnekaTask instances
           double step = 0.01;
           double min = -2.0;
           double max = 2.0;
           while (min <= max)
           {
               // Step 2. create a task instance
               MyTask task = new MyTask();
               task.X = min;
               // Step 3. wrap the task instance into a AnekaTask
               AnekaTask gt = new AnekaTask(task);
               // Step 4. Submit the execution
               app.ExecutionWorkUnit(gt);

               min += step;
           }
       }
       /// <summary>
       /// Program entry point.
       /// </summary>
       /// <param name="args">program arguments</param>
       private static AnekaApplication<AnekaTask,TaskManager>
               Setup(string[] args)
        { .... }
}
```

Listing 3 - Task creation and submission.

As it can be noticed, in order to use the Task Model we have to include the namespaces *Aneka.Tasks* and *Aneka.Entity*. In general, any application that is based on a specific programming model supported by Aneka relies on two different set of dependencies:

- Core APIs: these dependencies are common to all programming model and are constituted by the following three libraries:
    - *Aneka.dll* (Namespaces: *Aneka*, *Aneka.Entity, Aneka.Security*): core classes of the common API (i.e.: *WorkUnit*, *AnekaApplication*, *Configuration*);
    - *Aneka.Data.dll* (Namespaces: *Aneka.Data*, *Aneka.Data.Entity*): classes supporting file management (i.e.: *FileData*, *HostData*);
    - *Aneka.Util.dll* (Namespace: *Aneka*): support classes used by the common APIs.
- Programming Model APIs: these dependencies are specific to the programming used and in the case of the Task Model consist in the *Aneka.Tasks.dll*. In general the APIs of a programming model are organized into three different namespaces that separate the components required for executing that model:
    - *Aneka.[Model]* (assembly: *Aneka.[Model].dll*)
    - *Aneka.[Model].Scheduling* (assembly: *Aneka.[Model].Scheduling.dll*)
    - *Aneka.[Model].Execution* (assembly: *Aneka.[Model].Execution.dll*)

  In order to develop an application based on a specific programming model it is necessary to reference only the first assembly, which contains the core elements of the model. The other two components are only required by the Aneka container for respecively coordinate the execution of the model and execute their specific units of computation.

All these libraries are located into the *[Aneka Installation Directory]\bin* directory and need to be copied into the directory of the TaskDemo project to successfully build the application.

It is important to notice that the *MyTask* class needs to be marked *Serializable* in order to be used as an *ITask* instance. Once the *MyTask* class has been defined, the submission of tasks to Aneka is really simple: we need just to configure the tasks with the required data for the computation, wrap them into a *AnekaTask* instance and submit it by using the *AnekaApplication* instance. On the grid node the Execute method will be invoked and after its execution the task will be sent back to the user. In order to submit tasks it is important to correctly setup the *AnekaApplication* instance.

Listing 3 contains almost everything needed to submit tasks to Aneka[2]. Once we have set up a Visual Studio project referencing the previous mentioned assemblies it is only necessary to write the content of Listing 3 and compile it as a console application. We can also build the application from the command line by using the following command:

```
csc /r:System.dll /r:Aneka.Tasks.dll /r:Aneka.dll /r:Aneka.Data.dll
/r:Aneka.Util.dll /main:Aneka.Examples.Program /out:MyTaskDemo.exe
/target:exe MyTaskDemo.cs
```

Let us inspect the previous command line. It tells the C# compiler to reference the previously mentioned assembly, to produce an executable whose name is *MyTaskDemo.exe* by compiling the *MyTaskDemo.cs* file.

3.3 AnekaApplication Configuration and Management

So far we just saw how to create tasks and submit tasks to Aneka by means of the *AnekaApplication* class. We did not say anything about how to configure a *AnekaApplication* instance, monitor its execution, and getting the results back. These operations are fundamental when working with the Task Model because the *AnekaTask* class does not provide any feedback during execution, but the results of the computation are obtained by interacting with the *AnekaApplication* instance.

3.3.1 Getting Information from AnekaApplication

The *AnekaApplication* class represents the gateway to a given Aneka grid. In order to interact with an installation of Aneka it is necessary to create and configure a *AnekaApplication* instance. This class provides:

- properties for querying the application status and retrieving general informations;
- methods for submitting jobs and controlling the execution;
- events for monitoring the execution and getting the results of the computation.

Listing 4 provides a complete reference of the *AnekaApplication* public interface.

```
namespace Aneka.Entity
{
    /// <summary>
    /// Class AnekaApplication. Represents an application which can be executed on
    /// Aneka grids. It manages a collection of work units which are submitted to
    /// the grid for execution.
    /// </summary>
```

2 For now we just skip the definition of the Setup method and we assume it as provided.

```
    /// <typeparam name="W">work unit specific type</typeparam>
    /// <typeparam name="M">application manager</typeparam>
    public class AnekaApplication<W, M>
                where W : WorkUnit
                where M : IApplicationManager, new()
    {
        #region Properties
        /// <summary>
        /// Gets the home directory used by the application to store files.
        /// </summary>
        public string Home { get { ... } }
        /// <summary>
        /// Gets the application state.
        /// </summary>
        public ApplicationState State { get { ... } }
        /// <summary>
        /// Gets the underlying application manager.
        /// </summary>
        public M ApplicationManager { get { ... } }
        /// <summary>
        /// Gets true if the application is finished.
        /// </summary>
        public bool Finished { get { ... } }
        /// <summary>
        /// Gets, sets the user credential
        /// required to execute the application.
        /// </summary>
        public ICredential UserCredential { get { ... } set { ... } }
        /// <summary>
        /// Gets the application identifier.
        /// </summary>
        public string Id { get { ... } }
        /// <summary>
        /// Gets the application creation
        /// date and time.
        /// </summary>
        public DateTime CreatedDateTime { get { ... } }
        /// <summary>
        /// Gets, sets the application display name.
        /// The application display name is the one
        /// that is used for display purposes.
        /// </summary>
        public string DisplayName { get { ... } set { ... } }
        #endregion

        #region Constructors
        /// <summary>
        /// Creates a AnekaApplication instance with
        /// the given user credentials.
        /// </summary>
        /// <param name="configuration">configuration</param>
        public AnekaApplication(Configuration configuration) { ... }
        /// <summary>
        /// Creates a AnekaApplication instance with the
        /// given display name and configuration.
        /// </summary>
        /// <param name="displayName">application name</param>
        /// <param name="configuration">configuration</param>
        public AnekaApplication(string displayName, Configuration configuration)
        { ... }
        #endregion

        #region Events
        /// <summary>
        /// Fires whenever a work unit belonging to the application
        /// changes status to finished.
        /// </summary>
        public event EventHandler<WorkUnitEventArgs<W>> WorkUnitFinished;
        /// <summary>
        /// Fires whenever a work unit belonging to the application
        /// changes status to failed.
        /// </summary>
        public event EventHandler<WorkUnitEventArgs<W>> WorkUnitFailed;
        /// <summary>
        /// Fires whenever a work unit belonging to the application
        /// changes status to failed.
        /// </summary>
        public event EventHandler<WorkUnitEventArgs<W>> WorkUnitAborted;
        /// <summary>
        /// Fires when application execution is completed.
        /// </summary>
        public event EventHandler<ApplicationEventArgs> ApplicationFinished;
        #endregion

        #region Work Units Management
        /// <summary>
        /// Adds a work unit to the application.
        /// </summary>
        /// <param name="workUnit">work unit</param>
        public void AddWorkUnit(W workUnit) { ... }
        /// <summary>
        /// Deletes a work unit from the list of the
        /// work units of the application.
        /// </summary>
        /// <parameter name="workUnit">work unit</param>
        public void DeleteWorkUnit(W workUnit) { ... }
        /// <summary>
        /// Gets the work unit corresponding to
        /// the given index.
        /// <summary>
        /// <param name="index">work unit string identifier</param>
        public W this[string index] { get { ... } }
        /// <summary>
        /// Adds the given dependencies to the dependencies
        /// statically inferred by using reflection.
        /// <summary>
        /// <param name="dependencies">list of dynamic dependent modules</param>
        public void ProvideDynamicDependencies(IList<ModuleDependency> dependencies)
        { ... }
        #endregion

        #region Execution
        /// <summary>
        /// Submits the application to the grid.
        /// </summary>
        public void SubmitExecution() { ... }
        /// <summary>
        /// Execute a work unit while application is running...
        /// </summary>
        /// <param name="workUnit">work unit</param>
        public void ExecuteWorkUnit(W workUnit) { ... }
        /// <summary>
        /// Stop the single work unit.
        /// </summary>
        /// <param name="workUnitId">work unit identifier</param>
        public void StopWorkUnit(string workUnitId) { ... }
        /// <summary>
        /// Stop the execution of application.
        /// </summary>
        public void StopExecution() { ... }
        #endregion

        #region File Management
        /// <summary>
        /// Adds the selected file to the list of files shared among
        /// all the work units.
        /// </summary>
        /// <param name="filePath">path to the file to add</param>
        public void AddSharedFile(string filePath) { ... }
        /// <summary>
        /// Removes the selected file from the list of files shared among all the
        /// work units.
        /// </summary>
        /// <param name="filePath">path to the file to remove</param>
        public void RemoveSharedFile(string filePath) { ... }
        #endregion
    }
```

```
}
```

Listing 4 - AnekaApplication class public interface.

The first thing that can be noticed that the *AnekaApplication* class is a generic type. And that it is specialized during instantiation with the specific execution model we want to use. We need to provide two different elements:

- *WorkUnit* specific type **W**;
- *IApplicationManager* specific type **M**;

These elements are strongly coupled and cannot be chosen separately. The mapping is the following:

- **W**: *AnekaTask* then **M**: *TaskManager*;
- **W**: Aneka*Thread* then **M**: *ThreadManager*;
- ...

The *AnekaApplication* class exposes some properties that are useful to monitor the status of the application and provide information about it. For example we can check the boolean property *Finished* to know whether the application is terminated or not. For a more detailed information about the execution status the can check the *State* property. The state property is defined as follows:

```
namespace Aneka.Entity
{
    /// <summary>
    /// Enum ApplicationState. Enumerates the different states through which the
    /// application transit.
    /// </summary>
    [Serializable]
    [Flags]
    public enum ApplicationState
    {
        /// <summary>
        /// Initial state of the application.
        /// </summary>
        UNSUBMITTED     =     1,
        /// <summary>
        /// This flag is set when the application is submitted [transient state].
        /// </summary>
        SUBMITTED       =     2,
        /// <summary>
        /// This flag is set when the application is running.
        /// </summary>
        RUNNING         =     4,
        /// <summary>
        /// This flag is set when the application completes.
        /// </summary>
        FINISHED        =     8,
        /// <summary>
        /// If this flag is set an error is occurred.
        /// </summary>
        ERROR           =     16
    }
}
```

Listing 5 - ApplicationState enumeration.

Other useful informations are:

- *Id:* automatically generated unique identifier for the application.
- *DisplayName:* gets and sets the name of the application. This information is only used for recording purposes and visualization.
- *CreatedDateTime:* gets the creation date of the application.
- *UserCredential*: gets and sets the user credential used to authenticate the client application to Aneka.

The *AnekaApplication* class provides also access to the underlying application manager by means of the *ApplicationManager* property. Only sophisticated applications that require a finer control on the task execution and monitoring require the user to interact with the application manager. These issues go beyond the scope of this tutorial and will not be addressed anymore.

3.3.2 Application Configuration

In order to create run the Task Model we need to create the *AnekaApplication* instance first. While it is possible to omit the display name of the application, we need to provide a *Configuration* object which tunes the behavior of the application being created.

```
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>
```

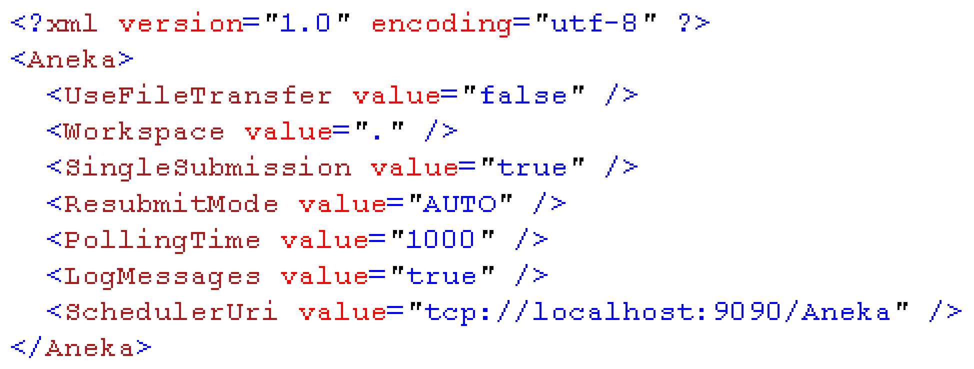
```
        public PropertyGroup AddGroup(PropertyGroup group) { ... }
        /// <summary>
        /// Removes the group of properties corresponding to the given name from the
        /// configuration if present.
        /// </summary>
        /// <param name="groupName">name of the property group 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 6 - Configuration class public interface.

Listing 6 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 like the following:



```
<?xml version="1.0" encoding="utf-8" ?>
<Aneka>
  <UseFileTransfer value="false" />
  <Workspace value="." />
  <SingleSubmission value="true" />
  <ResubmitMode value="AUTO" />
  <PollingTime value="1000" />
  <LogMessages value="true" />
  <SchedulerUri value="tcp://localhost:9090/Aneka" />
</Aneka>
```

Figure 2. Aneka Configuration File

There are few parameters which are worth to spend some more word for. These are: *SingleSubmission*, *ResubmitMode*, and *UserCredential*.

SingleSubmission influences the check made by the application to verify its completion status. When *SingleSubmission* is set the terminates its execution when the list of task submitted to the grid is empty and all the tasks have been returned and terminated their execution successfully. In case the *SingleSubmission* parameter is set to false,it is responsibility of the user code to detect the termination condition of the application and communicate it to *AnekaApplication* instance. In many cases there is no need to dynamically submit the jobs, but they can be provided all at once in a single submission as shown in *Listing 7.* From here the name *SingleSubmission*.

NOTE: In this tutorial we will not discuss the use of the file transfer support infrastructure. This feature allows to add additional files to the entire *AnekaApplication* or the single *WorkUnit*. The system will automatically transfer the files and retrieve the output files in a complete transparent manner. For more information about this feature it is possible to have a look at the documentation of the *Aneka.Data* and *Aneka.Data.Entity* documentation.

```
/// <summary>
/// Program entry point.
/// </summary>
/// <param name="args">program arguments</param>
public static void Main(string[] args)
{
    AnekaApplication<AnekaTask,TaskManager> app = Setup(args);
    // create task instances and wrap them
    // into AnekaTask instances
    double step = 0.0001;
    double min = -2.0;
    double max = 2.0;
    while (min <= max)
    {
        // Step 2. create a task instance
        MyTask task = new MyTask();
        task.X = min;
        // Step 3. wrap the task instance into a AnekaTask
        AnekaTask gt = new AnekaTask(task);
        // Step 4. instead of directly submitting
        // the task we add it to the application
        // local queue.
        app.AddWorkUnit(gt);

        min += step;
      }
      // Step 5. we submit all the tasks at once
      // this method simply iterates on the local
      // queue and submit the tasks.
      app.SubmitExecution();
```

```
        }
    }
}
```

Listing 7 - Task creation and submission in SingleSubmission mode.

The while loop shown in *Listing 7* is a possible solution for submitting tasks to Aneka: this is a single submission scenario, which is really useful for independent bag of tasks applications. In this case *SingleSubmission* can be set to true and the *AnekaApplication* class can automatically detect the termination of the execution without the user intervention.

SingleSubmission and Application Termination

SingleSubmission set to true, covers very simple scenarios in which the tasks are submitted only at once. In this case all the task to submit are added to the *AnekaApplication* instance and the *AnekaApplication.SubmitExecution()* method is invoked. This pattern implies that **there will be no further submissions of jobs**. If this is not the case, it is better to keep the *SingleSubmission* parameter set to false.

If we set the *SingleSubmission* to false and use method *AnekaApplication.SubmitWorkUnit(...)* in conjunction with the method *AnekaApplication.SubmitExecution()* there are chances that some task just simply get lost.

ResubmitMode allows users to tune the *AnekaApplication* behaviour. This parameters specify which resubmission strategy to adopt when a task fail. You can specify *ResubmitMode.AUTO* or *ResubmitMode.MANUAL*. In the first case when a task fails Aneka will take the responsibility of resubmitting the task, in the second case it is responsibility of the user to resubmit the task. More details will be given in Section 3.3.4.

UserCredential allows to specify the authentication credentials used by the client application to connect to Aneka and submit tasks. This property if of type *ICredential* and in order to provide a concrete instance for this property we can use the *UserCredential* class that is displayed in Listing 8 and allows user to be authenticated by using providing a simple user name and password. The class also exposes a list of groups that specify the roles that the user can play in the system and the operations that he or she can perform.

```
using System;
namespace Aneka.Security
{
    /// <summary>
    /// Interface ICredential. It provides the interface
    /// for all types of credential supported by the system.
    /// </summary>
    public interface ICredential
    {
        /// <summary>
        /// Gets, sets the unique identifier
        /// for the user credential.
        /// </summary>
        object Id { get; set; }
        /// <summary>
        /// Checks whether this instance is equal to the given credential.
        /// </summary>
        /// <param name="credential">credential</param>
        /// <returns>true if equal, false otherwise</returns>
        bool AreEqual(ICredential credential);
        /// <summary>
        /// Converts the credential data to a byte array.
        /// </summary>
        /// <returns>byte array containing the serialized
        /// credential instance</returns>
        byte[] ToByte();
        /// <summary>
        /// Constructs a credential instance from a byte stream.
        /// </summary>
        /// <param name="byteData">array of byte containing the serialized data of
        /// the credential instance</param>
        /// <returns>credential instance</returns>
        ICredential FromByte(byte[] byteData);
    }

    /// <summary>
    /// Class UserCredentials. Contains the information
    /// about the user by storing the username, the password,
    /// and the number of groups the user belongs to.
    /// </summary>
    [Serializable]
    public class UserCredentials : ICredential, IXmlSerializable
    {
        /// <summary>
        /// Gets, sets the user name of the user.
        /// </summary>
        public string Username { get { ... } set { ... } }
        /// <summary>
        /// Gets, sets the password for the user.
        /// </summary>
        public string Password { get { ... } set { ... } }
        /// <summary>
        /// Gets, sets the groups the user belongs to.
        /// </summary>
        public List<string> Groups { get { ... } set { ... } }
        /// <summary>
        /// Gets, sets the full name of the user.
        /// </summary>
        public string FullName { get { ... } set { ... } }
        /// <summary>
        /// Gets, sets the description of the user.
        /// </summary>
        public string Description { get { ... } set { ... } }

        /// <summary>
        /// Creates an instance of the UserCredentials type with blank user name
        /// and password.
        /// </summary>
        public UserCredentials(): this("", "", new List<string>()) { ... }
        /// <summary>
        /// Creates an instance of the UserCredentials type
        /// with given user name and password.
        /// </summary>
        /// <param name="username">user name</param>
        /// <param name="password">password</param>
        public UserCredentials(string username, string password) :
            this(username, password, new List<string>())
        { ... }
        /// <summary>
        /// Creates an instance of the UserCredentials type with given user name,
        /// password, and set of groups the user belongs to.
        /// </summary>
        /// <param name="username">user name</param>
        /// <param name="password">password</param>
        /// <param name="grps">list of groups the user belongs to</param>
        public UserCredentials(string username, string password, List<string> grps)
        { ... }
        /// <summary>
        /// Creates an instance of the UserCredentials type with given user name,
        /// password, and full details.
        /// </summary>
        /// <param name="username">user name</param>
        /// <param name="password">password</param>
        /// <param name="fullname">list of groups the user belongs to</param>
        /// <param name="description">list of groups the user belongs to</param>
        public UserCredentials(string username, string password, string fullname,
                               string description)
        { ... }
```

```
        #region ICredential Members
        ...
        #endregion

        #region IXmlSerializable Members
        ...
        #endregion

    }
}
```

Listing 8 - ICredential interface, Role enumeration, and UserCredentials class.

In order to use this class we need to include the namespace *Aneka.Security* in the namespace declaration. There is no need to reference additional assemblies because this class if defined in the *Aneka.dll* library. The *ICredential* interface does not specify anything except a property *Id* and methods for saving the data into a byte array and restoring an instance from it. This gives an high degree of flexibility in implementing authentication techniques, the specific features of each security model supported by the system, are exposed in the classes that implement this interface, for example the *UserCredentials* class.

The *Configuration* class basically performs the task of reading the settings concerning Aneka in the configuration file and exposing them within the application. The programmatic use of this class is generally limited and restricted to very simple applications and the common scenario is to have a configuration file where the user can put its settings.

3.3.3 Monitoring: Getting Back Results

Job submission is only a phase of grid-based application development. After submitting jobs, we are generally interested in handling the outcomes of their executions. Another important issue is knowing whether jobs have been executing successfully or have failed. In other words we need to *monitor* the execution of the grid application.

The *AnekaApplication* class exposes a set of events that user can register with, and that provide an asynchronous feedback about the application execution. These events are:

- *WorkUnitFinished:* this event is raised when the client application receives back from the Aneka scheduler service a job which has terminated its execution.
- *WorkUnitFailed:* this event is raised when the client application receives back from the Aneka scheduler service a job which has failed, along with some information – if available - on the nature of the error occurred.

- *WorkUnitAborted*: this event is raised when the user programmatically stops the execution of a work unit already submitted to Aneka.
- *ApplicationFinished:* this event is raised by the client application as soon as the application termination condition is verified. This can be detected automatically or with the contribution of the user.

WorkUnitFinished, WorkUnitFinished, and *WorkUnitAborted* are of the same type: *EventHandler<WorkUnitEventArgs<W>>.* Listing 8 shows the *WorkUnitEventArgs* class and the enumeration *WorkUnitState.*

```
namespace Aneka.Entity
{
    /// <summary>
    /// Class WorkUnitEventArgs. Defines the information related to change status
    /// event of a WorkUnit. It basically shows the information on the WorkUnit
    /// instance, its status, and an exception if occurred during execution.
    /// </summary>
    [Serializable]
    public class WorkUnitEventArgs : EventArgs
    {
        /// <summary>
        /// Gets the work unit instance related to the event.
        /// </summary>
        public WorkUnit WorkUnit { get { ... } }

        // constructors...
     }
     /// <summary>
     /// Enum WorkUnitState. Defines the possible states of a WorkUnit.
     /// </summary>
     public enum WorkUnitState
     {
        /// <summary>
        /// The work unit has not been  started yet.
        /// </summary>
        Unstarted       = 0,
        /// <summary>
        /// The work unit is running on some node of the grid.
        /// </summary>
        Running         = 1,
        /// <summary>
        /// The work unit has been stopped.
        /// </summary>
        Stopped         = 2,
        /// <summary>
        /// The work unit is suspended (Not used at the moment).
        /// </summary>
        Suspended       = 3,
        /// <summary>
        /// The work unit has started. (Used in the Thread Model)
        /// </summary>
        Started         = 4,
        /// <summary>
        /// The work unit has been queued.
        /// </summary>
        Queued          = 5,
        /// <summary>
        /// The work unit failed.
        /// </summary>
        Failed          = 6,
        /// <summary>
        /// The work unit has been rejected.
        /// </summary>
        Rejected        = 7
        /// <summary>
        /// The work unit is waiting for input files to be moved on the server.
        /// </summary>
        StagingIn       = 8,
        /// <summary>
        /// The work unit has completed its execution and it is waiting for its
        /// files to be moved on the client.
        /// </summary>
        StagingOut      = 9,
        /// <summary>
        /// The instance has been aborted by the user.
        /// </summary>
        Aborted         = 10,
        /// <summary>
        /// The instance has terminated its execution successfully and all its
        /// depending output files have been downloaded.
        /// </summary>
        Completed       = 11,

        /// <summary>
        /// The instance was running and for some reason has been terminated by
        /// the infrastructure and rescheduled.
        /// </summary>
        ReScheduled     = 12

     }
}
```

Listing 9 - Configuration class public interface and WorkUnitState enumeration.

The *WorkUnitEventArgs* class provides only one property: *WorkUnit*. Once we get a reference to the *WorkUnit* we can query its *State* property to get information about the work unit status according to the values of the enumeration *WorkUnitState*.

When the event *WorkUnitFailed* is raised the corresponding the user should expect only the values *Failed*, *Rejected*, and *Stopped.* In this case it is possible to look at the *Exception* property of the *WorkUnit* instance. This property can be helpful in discovering the origin of the failure. There could be different causes originating this event:

- *The computing node on which the task was executed failed*. In this case Aneka will send back a *WorkUnitFailed* event by setting the *Status* property of the *WorkUnit* to *Failed* and by setting the *Exception* property to *System.Exception* with “Resource Failure” as message. This condition raises the WorkUnitFailed event if and only if the *ResubmitMode* on the *WorkUnit* instance has been set to *ResubmitMode.MANUAL*.
- *The task execution has been rejected.* This case only applies in case of using advance reservation. Advance reservation allows clients to reserve a future time slice for executing a work unit. This is an advanced feature and will not be discussed in this tutorial. The only thing important to mention is that in this case the *Status* property of the *WorkUnit* instance will be set to *Rejected*.
- *The task execution failed because of an exception in the user code.* This is the most common case. If the task fails because of an exception in the user code the Status property is set to *WorkUnitState.Failed* and the *Exception* property will contain the remote exception which generated the failure.

Handling Properly the WorkUnitFailed Event

It is always wise to perform null value checks on the *WorkUnit* and the *Exception* property of the *WorkUnit* instance, in case of *WorkUnitFailed* events. Since something has gone wrong on the remote computation node and we are not sure about the cause, we cannot be sure that all the data of the failure have been collected.

While we are always assured that the *WorkUnit* property of the event argument is not null, but we cannot ensure this for its *Exception* property.

The *WorkUnitAborted* event is raised only in case the user actively stops the execution of a work unit that has been already submitted. This event is useful when multiple software components are monitoring the activity of the application. In this scenario one software component can stop the execution of the work unit and the other one can get notified of this action.

In case of *WorkUnitFinished* the state property will then be set to *Stopped*.

This is the simplest case: we can then access the *WorkUnit* property - that in the case of task model would be of type *AnekaTask* – to access all the data of the *WorkUnit* and getting back the results.

The case of the *ApplicationFinished* event is simpler. As shown in Listing 10, the *ApplicationEventArgs* class only contains one property that is the duration of the entire execution of the application.

```
namespace Aneka.Entity
{
    /// <summary>
    /// Class ApplicationEventArgs. Event argument class providing information about
    /// the execution of a grid application.
    /// </summary>
    [Serializable]
    public class ApplicationEventArgs : EventArgs
    {
        /// <summary>
        /// Gets the duration of the execution of the grid application.
        /// </summary>
        public TimeSpan Duration { get { ... } }

        ...
        // constructors...
    }
}
```

Listing 10 - ApplicationEventArgs class public interface.

It is now possible to write the user code that will collect the results. Basically what we need to do is to register handlers with these events and process the data inside the event handlers. We are not interested in handling the *WorkUnitAborted* event because there is only one component monitoring the life cycle of the application. In order to show how this can be done we will extend the previous example (See Listing 3) by adding the code to obtain the value of the Gaussian distribution and saving it into an array. We will also handle the case of tasks failed. Here is the list of the steps to perform:

1. Define a static *Dictionary<double,double>* called *samples* and instantiate it at the beginning of the *Main* method.
2. Define a static integer field called failed and set it to 0 at the beginning of the *Main* method.

```
// File: MyTaskDemo.cs

using Aneka.Entity;
using Aneka.Tasks;
```

```
// adding some useful namespaces
using System;
using System.Collection.Generics;

namespace Aneka.Examples.TaskDemo
{
    ....
    class MyTaskDemo
    {
        /// <summary>
        /// failed task counter
        /// </summary>
        private static int failed;
        /// <summary>
        /// Dictionary containing sampled data.
        /// </summary>
        private static Dictionary<double,double> samples;
        /// <summary>
        /// Program entry point.
        /// </summary>
        /// <param name="args">program arguments</param>
        public static void Main(string[] args)
        {
            samples = new Dictionary<double,double>();
            failed = 0;
            .....
        }
    }
}
```

3. For each iteration of the for loop put an entry into the dictionary samples where the key is the value of *x* of the current iteration, and the value is the constant *double.NaN.*

```
// File: MyTaskDemo.cs

using Aneka.Entity;
using Aneka.Tasks;

namespace Aneka.Examples.TaskDemo
{

     ....
           while (min <= max)
           {
               // Step 2. create a task instance
               MyTask task = new MyTask();
               task.X = min;
               // Step 3. Wrap into AnekaTask
```

```
                AnekaTask gt = new AnekaTask(task);
                // Step 4. Submit the execution
                app.ExecuteWorkUnit(gt);

                   // map key to double.NaN
                   samples[task.X] = double.NaN;

                min += step;
            }
     ....
}
```

4. Define a static handler for the *WorkUnitFinished* event and register it with the event (*OnWorkUnitFinished).*
5. Define a static handler for the *WorkUnitFailed* event and register it with the event (*OnWorkUnitFailed*).
6. Define a static handler for the *ApplicatioFinished* event and register it with the event (*OnApplicationFinished*).

```
// File: MyTaskDemo.cs
    ....

            /// <summary>
            /// Program entry point.
            /// </summary>
            /// <param name="args">program arguments</param>
            public static void Main(string[] args)
            {
                  samples = new Dictionary<double,double>();
                  failed = 0;
                  AnekaApplication<AnekaTask,TaskManager> app
                                                        = Setup(args);
                  // registering with the WorkUnitFinished
                  // event
                  app.WorkUnitFinished +=
                    new EventHandler<WorkUnitEventArgs<AnekaTask>>
                    (OnWorkUnitFinished);
                  // registering with the WorkUnitFailed
                  // event
                  app.WorkUnitFailed +=
                    new EventHandler<WorkUnitEventArgs<AnekaTask>>
                    (OnWorkUnitFailed);
                  // registering with the ApplicationFinished
                  // event
                  app.ApplicationFinished +=
                    new EventHandler<ApplicationEventArgs>
                    (OnApplicationFinished);
                  .....
            }
            ....
      }
}
```

7. Inside the method *OnWorkUnitFinished* defined write the code required to unwrap the *MyTask* instance from the *WorkUnit* property and to replace the *double.NaN* value with the value of the Result property, for the corresponding value of *X*.

```
// File: MyTaskDemo.cs
    ....

            /// <summary>
            /// Handles the WorkUnitFinished event.
            /// </summary>
            /// <param name="sender">event source</param>
            /// <param name="args">event arguments</param>
            public static void OnWorkUnitFinished(object sender,
                                                  WorkUnitEventArgs args)
            {
                  MyTask task = args.WorkUnit.UserTask as MyTask;
                  lock(samples)
                  {
                        samples[task.X] = task.Result;
                  }
            }
     ....
```

8. Inside the method *OnWorkUnitFailed* write the code to increment the *failed* static field of one unit.

```
// File: MyTaskDemo.cs
    ....

            /// <summary>
            /// Handles the WorkUnitFailed event.
            /// </summary>
            /// <param name="sender">event source</param>
            /// <param name="args">event arguments</param>
            public static void OnWorkUnitFailed(object sender,
                                   WorkUnitEventArgs<AnekaTask> args)
            {
                  Threading.Interlock.Increment(failed);
            }
     ....
```

9. Inside the method *OnApplicationFinished* write the code for dumping the data of the dictionary and the number of task failed.

```
// File: MyTaskDemo.cs
    ....

            /// <summary>
            /// Handles the ApplicationFinished event.
            /// </summary>
            /// <param name="sender">event source</param>
            /// <param name="args">event arguments</param>
            public static void OnApplicationFinished(object sender,
                                                     ApplicationEventArgs args)
            {
                  // we do not need to lock anymore
                  // the samples dictionary because the
                  // asynchronous events are finished then
                  // there is no risk of races.
                  Console.WriteLine("Results");
                  foreach(KeyValuePair<double,double> sample in samples)
                  {
                        Console.WriteLine("{0}\t{1}",
                                          sample.Key,
                                          sample.Value);
                  }
                  Console.WriteLine("Tasks Failed: " + failed);
            }
    ....
```

Writing the code for monitoring the application has been very straightforward. The only aspect that worth a further discussion is the synchronization management.

As previously said events are asynchronous because they are generated from the execution stack of another process. Since we cannot make any assumption on the timing of these events we cannot be sure they will not overlap. Indeed, it is frequent that multiple event calls to the *WorkUnitFinished* are run concurrently. Since these calls operate on the same data structure it is necessary to lock before invoking any operation on it. The same happens for the increment of the *failed* field. Conversely, while executing the handlers registered with the *ApplicationFinished* event, there are no concurrent calls then the synchronization code is not required.

We can observe that the synchronization strategy presented in this sample is really trivial. For more complex scenarios this infrastructure cannot be enough and a more sophisticated strategy involving semaphores can be required.

3.3.4 Application Control and Task Resubmission

As we have already said, The *AnekaApplication* class provides methods for controlling its the execution on Aneka. We already have seen how to submit tasks by using two different strategies. And how these strategies affect the evaluation of the stop condition of the grid application. But we do not have shown how the user can programmatically stop the execution of the application. This is accomplished by the *AnekaApplication.StopExecution()* method.

By calling *StopExecution()* the user terminates the execution of the grid application on Aneka and aborts all the tasks still running belonging to the application. This method represents the only way for the user ensure the termination of the *AnekaApplication* when the *SingleSubmission* property is set to false. Setting the *SingleSubmission* property to false is not really rare, indeed it is requirement when the user wants to resubmit tasks that have failed their execution and this implies the responsibility of communicating to the *AnekaApplication* instance when the application is terminated. In this case, it is necessary to keep trace of the execution of tasks in smarter way than the one presented in the previous section. In this section we will show how to handle task resubmission and programmatically control the termination of the grid application by using the *StopExecution()* method.

Task resubmission is simply performing by collecting the information of the task failed by registering an handler for the WorkUnitFailed event and, after having collected all the information about the failed task, creating a new instance of the *AnekaTask* class and resubmit it by calling the AnekaApplication.ExecuteWorkUnit(...) method. This action actually submit a new task with the same configuration of the failed one.

As pointed out in section 3.3.2, Aneka gives you the flexibility to control the task resubmission behavior by the *ResubmitMode* property in the *Configuration* class. Listing 11 shows all the possible values of *ResubmitMode* enumeration.

```
namespace Aneka.Entity
{
     /// <summary>
     /// Enum ResubmitMode. Enumerates the strategies available for handling work
     /// unit resubmission on failure.
     /// </summary>
     public enum ResubmitMode
     {
         /// <summary>
         /// Aneka will automatically handle task resubmission.
         /// </summary>
         AUTO            = 0,
         /// <summary>
         /// Aneka will not handle task resubmission on node failure and will fire
         /// the WorkUnitFailed event.
         /// </summary>
         MANUAL          = 1,
         /// <summary>
```

```
        /// Inherit the value defined at configuration level.
        /// </summary>
        INHERIT         = 2,
    }
}
```

Listing 11 - ResubmitMode enumeration.

The default configuration for any *AnekaApplication* instance is *ResubmitMode.AUTO*. As previously said this means that Aneka will automatically handle resubmission on task failure. It is also possible two specify the *ResubmitMode.MANUAL*, in this case Aneka will not handle task resubmission and will fire the event *WorkUnitFailed* for each work unit that does not complete successfully. It is important to notice that **only** in case *ResubmitMode* is set to *MANUAL* the client will be notified of task failure.

NOTE: It is important to notice that the value of ResubmitMode only influences the behavior of Aneka **when a computing node fails**. A work unit that fails because of the abnormal termination of the user task (such as an Exception originating from the work unit user code) will always raise the *WorkUnitFailed* event, without any regard to the value of *ResubmitMode*. In fact, there is no point in try to resubmit the execution of the user code without any action, because that code will probably always fail.

These are the only two values allowed for the *ResubmitMode* property in the *Configuration* class. There is, however, a third value which is *ResumitMode.INHERIT*. Such option is the default one for the *ResubmitMode* property of *WorkUnit* class and tells the work unit instance to inherit the configuration setting from the Configuration instance of the application.

The possibility of having a general task resubmission behavior and of tuning this behavior for the single tasks allows a great degree of flexibility in defining custom resubmission strategies. For example it is possible to select the manual resubmission mode only for critical tasks and let Aneka take care about all the other tasks resubmission. This strategy can be useful when timing is an important issue and when there are dependencies among tasks handled by the client application. In this tutorial we will only show how to handle the manual resubmission of all the tasks.

Manual task resubmission requires the *SingleSubmission* set to false. Given that, when handling task resubmission on the client side **we first have to define a termination condition** for the grid application. It is, actually, our responsibility to call the *AnekaApplication.StopExecution()* method to communicate to the *AnekaApplication* class to stop its execution. The general rule of thumb is the following:

1. Identify a termination condition for your application.
2. Write the handler for the *WorkUnitFinished* event.
3. Inside the code of the handler check the termination condition and eventually call *AnekaApplication.Stop()*.

Things become more complicate when we have to handle task resubmission, in this case it could possible that we have to modify some parameters which compose the termination condition.

Now we will extend the previous example in order to handle the manual resubmission and we will show how the problem can be easily solved when we have a fixed number of task to execute. We will proceed by devising two strategies:

- *Strategy 1: Log Only.* This strategy simply logs the failure of the task makes and updates the termination condition.
- *Strategy 2: Full Care.* This strategy resubmits the failed tasks until all the tasks have completed successfully.

Since we have a fixed number of tasks we can easily trace the execution of tasks by using two integer variables: *completed* and *total*. The first one will be incremented by one unit at each task completion, while the second one will store the total number of task that we expect to complete. The algorithm used to detect the application termination condition is then really straightforward:

1. The application terminates its execution when *(total == completed)* holds true.
2. Before starting to tasks submission set *current* to 0, and *total* to the number of tasks composing out application.
3. Inside the *WorkUnitFinished* event handler:
    1. increment by one unit the value of *completed*;
    2. check whether *total* is equal to *completed*;
    3. if the two values are equal call *AnekaApplication.StopExecution()*;

We can finally register an handler with the *ApplicatioFinished* event to post process all the tasks.

<u>Implementing Strategy 1: Log Only.</u>

In this case, since we do not need to resubmit the tasks the number of total tasks that we expect to complete decreases by one unit at each task failure. Then in this strategy we have to properly handle this condition into the *WorkUnitFailed* event handler. The algorithm needs to be modified by introducing a step 4 as the following:

4. Inside the *WorkUnitFailed* event handler:

1. decrement by one unit the value of *total*;
2. check whether *total* is equal to *completed*;
3. if the two values are equal call *AnekaApplication.StopExecution()*;
4. eventually log the failure of the task (*failed* field).

We can observe that since we are modifying the same variables in two different event handlers we should use a single synchronization object that is expressively added to the application by defining a static object field *synchLock* and initializing it (just call *new object()*) into the *Main* method, before submitting the tasks. Inside the two event handlers wrap all the code into the a *lock(synchLock) { … }* block. This practice instead of using *Interlock.Increment(…)* and *Interlock.Decrement(…)* ensures that change to the counters and the check for the termination condition are executed atomically in each event handler.

Implementing Strategy 2: Full Care.

In this case, we do not have to make any change to the general algorithm but we have to handle the task resubmission on the client side. This can be easily done inside the *WorkUnitFailed* event handler. It is just necessary to:

1. create a new instance of the *AnekaTask* class;
2. configure it with the *UserTask* property of the failed task;
3. call the *AnekaApplication.ExecuteWorkUnit(…)* method by passing to it the newly created *AnekaTask* instance.

We just want to observe that either the *WorkUnit* property of the *WorkUnitStatusArgs* instance or the *UsertTask* property of the *WorkUnit* instance can be null. In this case we need to implement a local cache of all the submitted tasks in order to be able to resubmit the tasks.

Listing 12 shows the complete application with an implementation of both the two strategies.

```
// File: MyTaskDemo.cs
using System;
using System.Threading;
using System.Collections.Generic;

using Aneka.Entity;
using Aneka.Security;
using Aneka.Tasks;


namespace Aneka.Examples.TaskDemo
{
    /// <summary>
    /// Class MyTask. Simple task function wrapping
    /// the Gaussian normal distribution. It computes
    /// the value of a given point.
    /// </summary>
    [Serializable]
    public class MyTask : ITask
    {
        /// <summary>
        /// value where to calculate the  Gaussian normal distribution.
        /// </summary>
       private double x;
       /// <summary>
       /// Gets, sets the value where to calculate the Gaussian normal distribution.
       /// </summary>
       public double X  { get { return this.x; } set { this.x = value; } }
       /// <summary>
       /// value where to calculate the Gaussian normal distribution.
       /// </summary>
       private double result;
       /// <summary>
       /// Gets, sets the value where to calculate the Gaussian normal distribution.
       /// </summary>
       public double Result
       {
            get { return this.result; }
            set { this.result = value; }
       }

       /// <summary>
       /// Creates an instance of MyTask.
       /// </summary>
       public MyTask() {}

       #region ITask Members
       /// <summary>
       /// Evaluate the Gaussian normal distribution
       /// for the given value of x.
       /// </summary>
       public void Execute()
       {
            this.result = (1 / (Math.Sqrt(2* Math.PI))) *
                                      Math.Exp(- (this.x * this.x) / 2);
       }
       #endregion
    }

    /// <summary>
    /// Class MyTaskDemo. Simple Driver application that shows how to create tasks
    /// and submit them to the grid, getting back the results and handle task
```

```
    /// resubmission along with the proper synchronization.
    /// </summary>
    class MyTaskDemo
    {
       /// <summary>
       /// failed task counter
       /// </summary>
       private static int failed;
       /// <summary>
       /// completed task counter
       /// </summary>
       private static int completed;
       /// <summary>
       /// total number of tasks submitted
       /// </summary>
       private static int total;

       /// <summary>
       /// Dictionary containing sampled data
       /// </summary>
       private static Dictionary<double,double> samples;

       /// <summary>
       /// synchronization object
       /// </summary>
       private static object synchLock;
       /// <summary>
       /// sempahore used to wait for application termination
       /// </summary>
       private static AutoResetEvent semaphore;
       /// <summary>
       /// grid application instance
       /// </summary>
       private static AnekaApplication<AnekaTask,TaskManager> app;

       /// <summary>
       /// boolean flag indicating which task failure management strategy to use.
       /// If true the Log Only strategy will be applied, if false the Full Care
       /// strategy will be applied.
       /// </summary>
       private static bool bLogOnly = false;

       /// <summary>
       /// Program entry point.
       /// </summary>
       /// <param name="args">program arguments</param>
       public static void Main(string[] args)
       {
           Console.WriteLine("Setting Up Grid Application..");
```

```
            app = Setup(args);

            // create task instances and wrap them
            // into AnekaTask instances
            double step = 0.01;
            double min = -2.0;
            double max = 2.0;

            // initialize trace variables.
            total = 400; // (max – min) / step
            completed = 0;
            failed = 0;
            samples  = new Dictionary<double,double>();

            // initialize synchronization data.
            synchLock = new object();
            semaphore = new AutoResetEvent(false);

            // attach events to the grid application
            AttachEvents(app);
            Console.WriteLine("Submitting {0} tasks...", total);
            while (min <= max)
            {
                  // create a task instance
                  MyTask task = new MyTask();
                  task.X = min;
                  samples.Add(task.X, double.NaN);
                  // wrap the task instance into a AnekaTask
                  AnekaTask gt = new AnekaTask(task);
                  // submit the execution
                  app.ExecuteWorkUnit(gt);
                  min += step;
            }
            Console.WriteLine("Waiting for termination...");
            semaphore.WaitOne();
            Console.WriteLine("Application finished.");

        }

        #region Helper Methods
        /// <summary>
        /// AnekaApplication Setup helper method. Creates and
        /// configures the AnekaApplication instance.
        /// </summary>
        /// <param name="args">program arguments</param>
        private static AnekaApplication<AnekaTask,TaskManager> Setup(string[] args)
        {
             Configuration conf = Configuration.GetConfiguration();
            // ensure that SingleSubmission is set to false
            // and that ResubmitMode to MANUAL.
            conf.SingleSubmission = false;
            conf.ResubmitMode = ResubmitMode.MANUAL;

            conf.UserCredential = new UserCredential(user, pass);

            AnekaApplication<AnekaTask,TaskManager> app =
                  new AnekaApplication<AnekaTask,TaskManager>"MyTaskDemo", conf);

            // ensure that SingleSubmission is set to false
            if (args.Length == 1)
            {
                  bLogOnly = (args[0] == "LogOnly" ? true : false);
            }
            return app;
        }
        /// <summary>
        /// Attaches the events to the given instance of the AnekaApplication class.
        /// </summary>
        /// <param name="app">grid application</param>
        private static void AttachEvents(AnekaApplication<AnekaTask,TaskManager> app)
        {
            // registering with the WorkUnitFinished event
            app.WorkUnitFinished +=
                 new EventHandler<WorkUnitEventArgs<AnekaTask>>(OnWorkUnitFinished);
            // registering with the WorkUnitFinished event
            app.WorkUnitFailed +=
                 new EventHandler<WorkUnitEventArgs<AnekaTask>>(OnWorkUnitFailed);
            // registering with the ApplicationFinished event
            app.ApplicationFinished +=
                 new EventHandler<ApplicationEventArgs>(OnApplicationFinished);
        }
        /// <summary>
        /// Dumps the results to the console along with some information about the
        /// task failed and the tasks used.
        /// </summary>
        private static void ShowResults()
        {
            // we do not need to lock anymore the samples dictionary because the
            // asynchronous events are finished then there is no risk of races.

            Console.WriteLine("Results");
            foreach(KeyValuePair<double,double> sample    in samples)
            {
               Console.WriteLine("{0}\t{1}", sample.Key, sample.Value);
            }
            Console.WriteLine("Tasks Failed: " + failed);
            string strategy = bLogOnly ? "Log Only" : "Full Care";
            Console.WriteLine("Strategy Used: " + strategy);
        }
        #endregion

        #region Event Handler Methods
        /// <summary>
        /// Handles the WorkUnitFailed event.
        /// </summary>
        /// <param name="sender">event source</param>
        /// <param name="args">event arguments</param>
        public static void OnWorkUnitFailed(object sender,
                                            WorkUnitEventArgs<AnekaTask> args)
        {
            if (bLogOnly == true)
            {
                  // Log Only strategy: we have to simply record the failure and
                  // decrease the number of total task by one unit.
                  lock(synchLock)
                  {
                       total = total - 1;
                       // was this the last task?
                       if (total == completed)
                       {
                           app.StopExecution();
                       }
                       failed = failed + 1;
                  }
            }
            else
            {
                  // Full Care strategy: we have to resubmit the task. We can do
                  // this only if we have enough information to resubmit it
                  // otherwise we switch to the LogOnly strategy for this task.
                  AnekaTask submitted = args.WorkUnit;
                  if ((submitted != null) && (submitted.UserTask != null))
                  {
                       MyTask task = submitted.UserTask as MyTask;
                       AnekaTask gt = new AnekaTask(task);
                       app.ExecuteWorkUnit(gt);
                  }
                  else
                  {
                       // oops we have to use Log Only.
                       lock(synchLock)
                       {
                           total = total - 1;
                           // was this the last task?
                           if (total == completed)
                           {
                               app.StopExecution();
```

```
                            }
                            failed = failed + 1;
                        }

                   }
            }
        }
        /// <summary>
        /// Handles the WorkUnitFinished event.
        /// </summary>
        /// <param name="sender">event source</param>
        /// <param name="args">event arguments</param>
        public static void OnWorkUnitFinished (object sender,
                                               WorkUnitEventArgs<AnekaTask> args)
        {
             // unwrap the task data
             MyTask task = args.WorkUnit.UserTask as MyTask;
             lock(synchLock)
             {
                  // collect the result
                  samples[task.X] = task.Result;
                  // increment the counter
                  completed = completed + 1;
                  // was this the last?
                  if (total == completed)
                  {
                      app.StopExecution();
                  }
             }
       }
       /// <summary>
       /// Handles the ApplicationFinished event.
       /// </summary>
       /// <param name="sender">event source</param>
       /// <param name="args">event arguments</param>
       public static void OnApplicationFinished(object sender,
                                                ApplicationEventArgs args)
       {
             // display results
             ShowResults();
             // release the semaphore
             // in this way the main thread can terminate
             semaphore.Set();
       }
       #endregion
    }
}
```

Listing 12 - Task creation and submission.

We can notice that example uses an *AutoReset* event object to make the main application thread wait until all the tasks have been returned by Aneka and the results have been showed to the console. If we do not make the main thread wait the program terminates before all the tasks get back. This is due to the fact that when then main application thread reaches the end of its scope and terminates it sends an abort message to all dependent threads.

In real application this technique is really common but there could be cases in which it is not necessary.

Finally a possible improvement of this application is the introduction of a local cache for keeping track of the tasks more precisely.

4. Conclusions

In this tutorial we have introduced the Task Execution Model implemented in Aneka for running a set of independent tasks. Within the Task Execution Model a task is specific work unit that can be executed on a remote node. The Task Execution Model is the simplest and intuitive execution model in Aneka. It is suitable to execute jobs of legacy code or managed .NET code.

In order to define a specific task we need to create a type implementing the *ITask* interface. This interface exposes only one method that is Execute which is the method called on the remote node to carry out the execution of the task. In order to be able to submit tasks to Aneka, the type defined need to be serializable, and all the information it needs for its execution has to be bundled with it (and eventually be serializable as well). Aneka will take care of unpacking the grid on the remote node, executing it, repack it, and send it back to the client.

This tutorial has covered the following arguments:

- General notions about the Task Model.
- How to define a class that implements the *ITask* interface.
- How to create and configure the *AnekaTask* instance with a user defined task.
- How to create an *AnekaApplication* instance and configure it with Task Model.
- How to submit tasks for execution.
- How to monitor the life cycle of a Task Model application.
- How to handle task resubmission.

All these features have been demonstrated by developing the *TaskDemo* application from scratch.

This tutorial does not fully cover what can be done with the Task Model. In particular this tutorial did not explained how to use input, output, and shared files with the application and the tasks. For a more detailed information about this and other aspects the user can have a look at the APIs documentation.