MANJRASOFT PTY LTD

Developing Thread Model Applications

Aneka 3.0

Manjrasoft

5/24/2012

This tutorial describes the Aneka Thread Programming Model and explains how to create distributed applications based on it. It illustrates some examples provided with the Aneka distribution that are built on top the Thread Model. It provides a detailed step by step guide for users on how to create an application by using the Microsoft Visual Studio 2005 Development Environment. After having read this tutorial, the users will be able to develop their own application on top of the Aneka Thread Model.

Table of Contents

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 installed1 even if not strictly required.

2 Introduction

Aneka allows different kind of applications to be executed on the same 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
- Parameter Sweep Model

Each execution model is composed by three 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 collect the results. A view of the system is given in the figure below.

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.

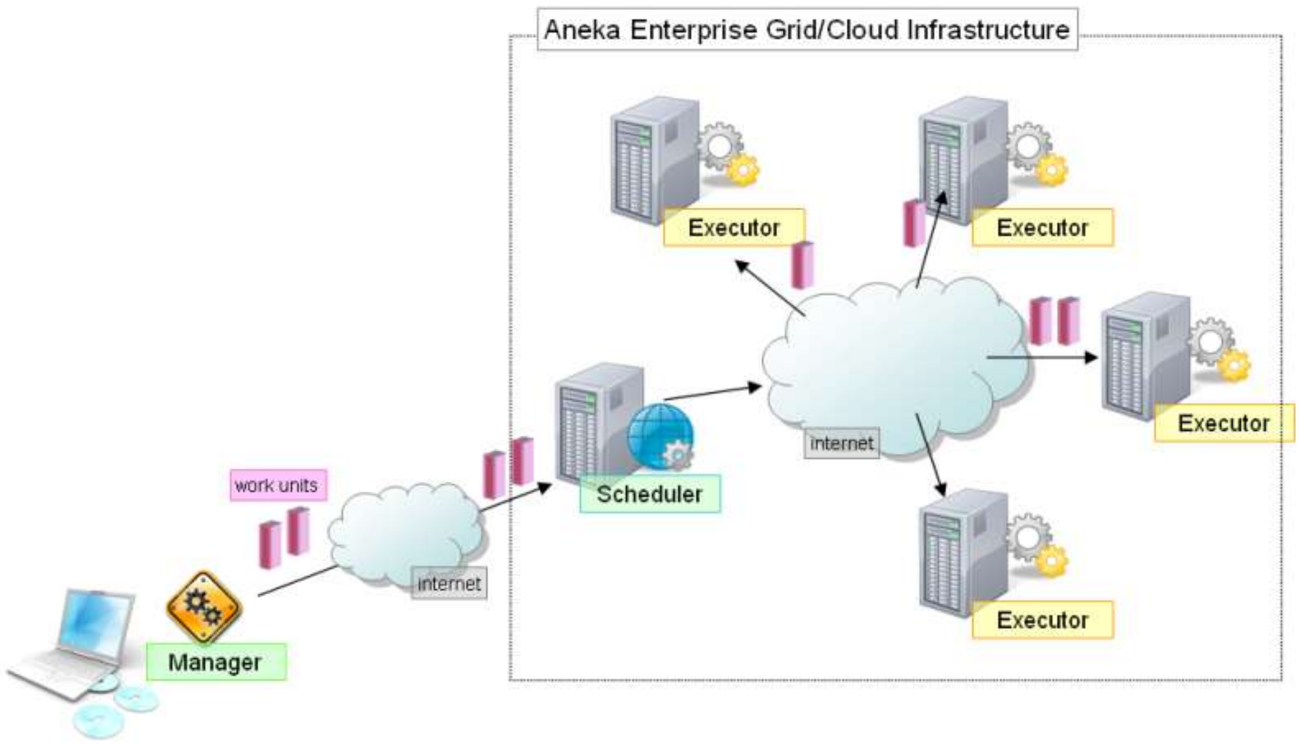


Figure 1 - System Components View.

Hence, for the Thread Model there will be a specific WorkUnit called AnekaThread, a Thread Scheduler, a Thread Executor, and a Thread 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 the users are required to know:

- how to define AnekaThread instances specific to the application that is being defined;
- how to create a AnekaApplication and starts the execution of threads;
- how to control the AnekaApplication and collect the results.

In the remainder of this tutorial will then concentrate on the Thread Model even if many of the concepts described can be applied to other execution models.

3 Thread Model

3.1 *Local vs Remote Threads*

The modern operating systems provide the abstractions of Process and Thread for defining the runtime profile of a software application. A Process is a software infrastructure that is used by the operating system to control the execution of an application. A Process generally contains one or more threads. A Thread is a sequence of instructions that can be executed in parallel with other instructions. When an application is running the operating system takes care of alternating their execution on the local machine. It is responsibility of the developer to create a consistent computation as a result of thread execution.

The Thread Model uses the same abstraction of for defining a sequence of instructions that can be remotely executed in parallel with other instructions. Hence, within the Thread Model an application is a collection of remotely executable threads. The Thread Model allows developers to virtualize the execution of a local multi-threaded application (developed with the .NET threading APIs) in an almost complete transparent manner. This model represents the right solution when developers want to port the execution of a .NET multi-threaded application on Aneka and still use the same way of controlling the execution of application flow, which is based on synchronization between threads.

Developers that are familiar with multi-threaded applications will find the Thread Model the most natural path to program distributed applications with Aneka. The transition between a .NET thread and an Aneka thread is almost transparent. In the following a sample application will be used to discuss how to use Aneka threads.

3.2 *Working with Threads*

Within the .NET threading model a thread is a represented by the *Thread* sealed class that can be configured with the method to execute through the *ThreadStart* class. The users activates a thread by calling the Start method on it and by using the APIs exposed by the Thread class it can:

- Check the status of the thread by using the *Thread.State* and the *Thread.IsAlive* properties.
- Control its execution by stopping it (*Thread.Abort*).
- Suspending and resuming their execution (*Thread.Suspend* and *Thread.Resume)*.
- Wait for its termination by calling *Thread.Join*.

The *Thread.Join*, *Thread.Suspend*, and *Thread.Resume* are the operations that allow developer to create basic synchronization patterns between threads. The Thread class provides additional APIs that cover:

- Thread affinity.
- Volatile read and write.
- Critical region management.
- Asynchronous operations.
- Stack management.

More complex and advanced synchronization pattern can be obtained by using other classes of the .NET threading APIs that does not have any reference to the *Thread* class.

In order to remotely execute a thread, the Thread Model provides a counterpart of the the Thread class: *AnekaThread*. The *AnekaThread* class represents the work unit in the Thread Model and exposes a subset of the APIs of *System.Threading.Thread*. It is possible to perform almost all the basic operations described before and the following table identifies the mappings between the two worlds.

.NET Threading API	Aneka Threading API
System.Threading	Aneka.Threading
Thread	AnekaThread
Thread.ManagedThreadId (int)	AnekaThread.Id (string, from WorkUnit)
Thread.Name	AnekaThread.Name (from WorkUnit)
Thread.ThreadState (ThreadState)	AnekaThread.State (WorkUnitState)
Thread.IsAlive	AnekaThread.IsAlive
Thread.IsRunning	AnekaThread.IsRunning (from WorkUnit)
Thread.IsBackground	[Not provided]
Thread.Priority	[Not provided]
Thread.IsThreadPoolThread	[Not provided]
Thread.Start	AnekaThread.Start
Thread.Abort	AnekaThread.Abort
Thread.Sleep	[Not provided]
Thread.Interrupt	[Not provided]
Thread.Suspend	[Not provided]
Thread.Resume	[Not provided]
....	[Not provided]

Table 1 - Local vs Remote Thread

The *AnekaThread* class implements the basic *Thread* operations but does not give any support for the advanced operations such as: critical region, stack, apartment, culture, and execution context management. Moreover, some basic operations have not been supported, these are:

- Thread Priority Management.
- Suspend, Resume, Sleep, and Interrupt.

The reason why these operations have not been supported is because the AnekaThread instances are remotely executed on a computation node that generally executes work units coming from different distributed applications. It is not possible to keep the resources of a computation node occupied with a *AnekaThread* instance that is sleeping, or suspended. For what concerns the priority Aneka does not provide any facility.

```
namespace Aneka.Threading
{
    /// <summary>
    /// Class AnekaThread. Represents the basic unit of work
    /// in the Thread Model. A AnekaThread instance is configured with a
    /// specific method to execute and its remote execution is
    /// started.
    /// </summary>
    public class AnekaThread : WorkUnit
    {
       /// <summary>
       /// Gets a boolean value indicating whether the AnekaThread instance
       /// instance is alive (not: Unstarted | Completed | Aborted |
       /// Failed)
       /// </summary>
       public bool IsAlive { get; }
       /// <summary>
       /// Gets the reflection and instance information on the method that is
       /// executed on the remote computation node.
       /// </summary>
       public RemoteMethodInfo TargetMethodInfo { get; }
       /// <summary>
       /// Gets the instance representing the target of the method invocation.
       /// </summary>
       public object Target { get; }
       /// <summary>
       /// Creates an instance of the AnekaThread.
       /// </summary>
       /// <param name="start">thread start method information.</param>
       /// <param name="application">grid application.</param>
       public AnekaThread(ThreadStart start,
                    AnekaApplication<AnekaThread, ThreadManager> application)
       { ... }
       /// <summary>
       /// Starts the execution of the AnekaThread.
       /// </summary>
       public void Start() { ... }
       /// <summary>
       /// Aborts the execution of the AnekaThread.
       /// </summary>
       public void Abort() { ... }
       /// <summary>
       /// Waits until the execution of AnekaThread is terminated.
       /// </summary>
       public void Join() { ... }
       /// <summary>
       /// Waits until the execution of AnekaThread is terminated.
       /// </summary>
```

```
        /// <param name="time">Maximum interval of time to wait.</param>
        public void Join(TimeSpan time) { ... }

    }
}
```

Listing 1 - AnekaThread class.

Listing 1 presents the public interface of the *AnekaThread* class. Other than the declaration of the basic operations for controlling the execution of a *AnekaThread*, the class exposes two interesting properties that provide information about the method executed remotely (*TargetMethodInfo*) and target of the invocation (*Target*).

In order to create a *AnekaThread* instance it is necessary to pass to the constructor two parameters: a *ThreadStart* object and a reference to the *AnekaApplication* instance that the threads belongs to. Once the thread has been created is *State* property is set to *WorkUnitState.Unstarted* and it is possible to access the information about the method that will be executed by the *TargetMethodInfo* property. This property extrapolates all the reflection information used to recreate the execution environment on the remote computation node. This property is of type *RemoteMethodInfo* whose interface is described in Listing 2.

```
namespace Aneka.Threading
{
    /// <summary>
    /// Class RemoteMethodInfo. Wraps all the required information for executing
    /// a method in a remote computation node.
    /// </summary>
    public sealed class RemoteMethodInfo
    {
        /// <summary>
        /// Gets the display name of the assembly containing the definition of
        /// the method to execute.
        /// </summary>
        public string AssemblyName { get; }
        /// <summary>
        /// Serialized information of the target of the method invocation.
        /// </summary>
        public byte[] ObjectInstance { get; }
        /// <summary>
        /// Flags used to invoke the method.
        /// </summary>
        public BindingFlags Flags { get; }
        /// <summary>
        /// Name of the method to invoke.
        /// </summary>
        public string TargetMethod { get; }
        ....
    }
}
```

Listing 2 - Class RemoteMethodInfo.

While the use of *TargetMethodInfo* is mostly internal, the *Target* property is of more interest for the user. This property exposes the updated value of the instance after the execution of the *AnekaThread* and provides a quick way for performing the mapping between the threads and the instances that are object of thread execution. When programming with the .NET threading API developers have to maintain this mapping in a specific data structure (list, hash-table, etc...) and it is their responsibility to keep it updated when the state of the thread changes. By using the Aneka threading APIs no additional code is required.

Static Methods

The .NET Framework allows the execution of static methods as entry points for Thread execution. While this feature makes perfectly sense in a local execution context, it becomes unclear in a distributed environment.

Threads running in the same application domain share a static context. As a result, the side effects of an execution can be captured into static variables and still be accessible to the user. Threads running in different application domains – and this is the case of AnekaThread instances – do not share a static context. This makes the use of static methods quite limited. For this reason the current implementation of the Thread Model does no support the remote execution of static methods.

The second parameter required by the *AnekaThread* constructor is the reference to the *AnekaApplication* class that groups all the instances belonging to the same application. A property common to all the programming models supported by Aneka is the concept of *application*. This can be generally described as a collection of related jobs that constitute a distiributed computation. All the programming models must provide a local view of the distributed application through a specific instance of the *AnekaApplication* class. *AnekaApplication<W,M>* is a generic class and need to be specialized with the concrete types that are related to the programming model implemented. In the case of the Thread Model we have that:

- W, which must inherit from *WorkUnit*, is *AnekaThread*.
- M, which must implement IApplicationManager, is ThreadManager.

These two generic parameters represent respectively the basic unit of computation of the model and the specific client manager used to handle the interaction with Aneka for the given programming model.

The specific tasks of the *AnekaApplication* are the interaction with Aneka and providing aggregate information on the execution of all the work units that it ows. Whereas in other programming models (see the *Task Model*) the *AnekaApplication* class plays a more concrete role, in the case of the *Thread Model* its role is mostly confined to the setup of the distributed application. The .NET Threading APIs do not have a corresponding application object and the execution flow of the application is mainly controlled by directly operating on the thread instances by calling the methods exposed by the *System.threading.Thread* class. In the case of Aneka the same approach has been maintained and developers can directly operate on *AnekaThread* instances once they have been created and assigned to a *AnekaApplication* instance.

The *AnekaThread* class provides all the required facilities to control its life cycle. Figure 2 depicts the life cycle of a *AnekaThread* instance. As soon as the instance is created it is in the *Unstarted* state. A call to *AnekaThread.Start()* makes it move into the *Started* state and causes the submission of the instance to Aneka. If the *AnekaThread* has some dependent files to be transferred it moves to *StagingIn* state until all the dependent files are transferred. The *AnekaThread* can then move directly to the *Running* state is any computing node is available or being queued, thus moving into the *Queued* state. As soon as execution completes if there are any dependent output files to be downloaded to the client the states is changed to *StatingOut* otherwise it is directly set to *Completed*. At any stage an exception or an error can occur that causes the *AnekaThread* instance to move into the *Failed* state. The user can also actively terminate the execution by calling *AnekaThread.Abort()* and this causes the *AnekaThread* instance to be stopped and its state to be set to *Aborted*.

NOTE: The diagram also shows the Rejected state. This state is related to the negotiation protocol and it is actually not active. AnekaThread instances can be allocated to specific slots for their execution that can be reserved exclusively. When starting an AnekaThread instance it is possible to associate to it a reservation identifier that will map the instance to the reserved slots. If this reservation identifier is not valid or expired the state of the instance becomes Rejected and it is not allowed to run. The diagram does not show to the ReScheduled state. This state is assumed when an AnekaThread instance is interrupted during execution and put in the scheduling queue gain. This could happen if the execution slot in which the instance was running has been pre-empted by a reserved WorkUnit instance.

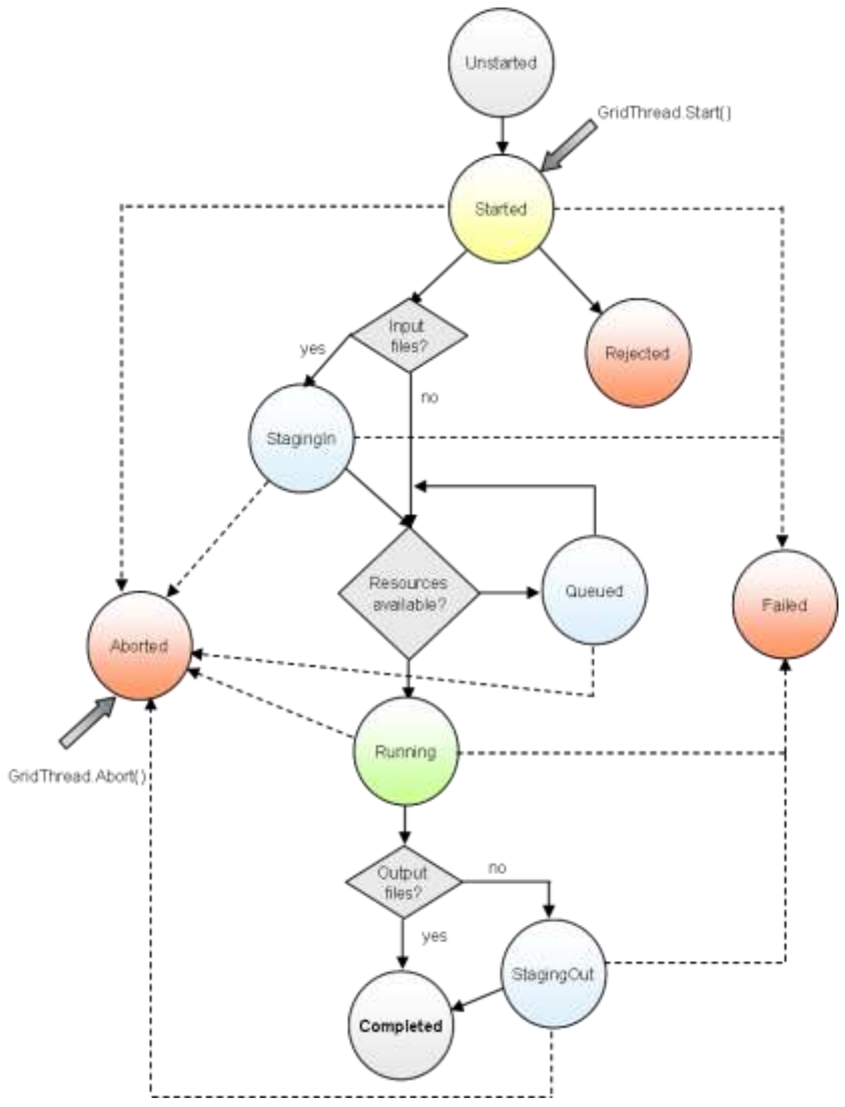


Figure 2 - AnekaThread instance state transitions (client view).

3.3 *Additional Considerations*

3.3.1 Serialization

Since *AnekaThread* instances are moved between different application domains they need to be serialized. The *AnekaThread* instance is declared *serializable*, but this does not guarantee that all *AnekaThread* instances created by the users will be serializable. In particular, since the *AnekaThread* is configured with a *ThreadStart* object referencing the instance that is the target of the method invocation, the type containing the method definition of the method need to be serializable too. The reason for this, is because the infrastrcture will serialize the local instance on which the method will be invoked and send it to the remote node.

In case the users provides a method that is not defined in a serializable type, the *AnekaThread* constructor throws an *ArgumentException* alerting the user that the selected method cannot be used to run a *AnekaThread* instance. This prevents the user from creating a work unit that will not run.

3.3.2 Thread Programming Model vs Common APIs

As pointed out in section 3.1 the Thread Model allow developers to completely controll the execution of the application by using the operations exposed by the *AnekaThread* class. Once the *AnekaApplication* instance has been properly set up there is no need to maintain a reference to it. The rationale behind this choice is that developers familiar with the .NET Threading APIs do not have the explicit concept of application but simply coordinate the execution of threads.

Since the Thread Model relies on the common APIs of the infrastructure, it takes advantage of the services these APIs offer and these services can be used by developers too. In this case the AnekaApplication class plays an important role in controlling the execution flow, since it allows to:

- Monitor the state of *AnekaThread* instances by using events:
  - AnekaApplication<W,M>.WorkUnitFailed
  - AnekaApplication<W,M>.WorkUnitFinished
  - GridApplicaiton<W,M>.WorkUnitAborted
- Programmatically control the execution of GirdThread instances:
  - AnekaApplication<W,M>.ExecuteWorkUnit(W)
  - AnekaApplication<W,M>.StopWorkUnit(W)
- Terminate the execution of the application:
  - AnekaApplication<W,M>.ApplicationFinished
  - AnekaApplication<W,M>.StopExecution

These APIs are available to all the models and allows developers to perform the basic operations required to manage the distributed application in a model independent fashion. For what concerns the Thread Model this seems to be unnatural even though can be useful some times. This tutorial will not explore further this option and the reader is suggested to look for the Task Model that naturally uses this APIs.

It is important to notice that the result of using the AnekaThread operations or the AnekaApplication operations is the same. The reason for this is that both of the two classes relies on the ThreadManager class for performing the requested operations.

4 Example: Distributed Warhol Filter.

In this section we will show how to use the *Thread Model* and the *AnekaThread* APIs to create a distributed image filter that performs the *Warhol Effect*. By developing this simple application the user will be able to:

- Create a AnekaApplication instance configured for the Thread Model.
- Customize the execution of the a AnekaApplication with a configuration file.
- Create and customize AnekaThread instances with user specific code.
- Submit and control the execution of AnekaThread instances.

This tutorial is not an exhaustive guide to the APIs provided with the Thread Model but it is a good start for developing applications based on distributed threads with Aneka.

4.1 *What is the Warhol Effect?*

There is no clear definition of what the Warhol Effect is but the effect transforms a given picture into another that resembles in style the following painting of Marylin Monroe made by the famous pop artist Andy Warhol (see Figure 3).

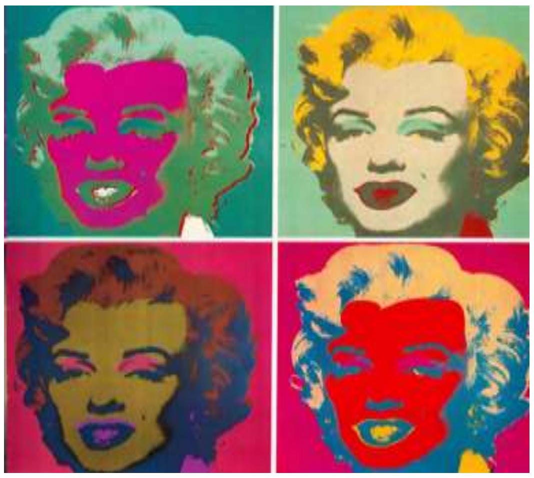


Figure 3 - Marylin Monroe prints (Andy Warhol).

Given the fact that the prints are made by a human without any specific algorithm it is quite difficult to automate the process and there are many attempts on the web that are trying to reproduce the same effect by means of a computer algorithm. Any filter available in the web produces a result whose similarity in style with Andy Warhol's paintings varies.

In order to produce a simple computer algorithm that perform the filter we will apply the following restrictions:

- The output image produced by the filter is has a color depth of four colors (many of the Warhol's paintings are basically made by using four colors).

- The colors of the original image are remapped and clustered into the new palette made of three colors according to their brightness.

- The output image will provide an image that is two times the size of the original one and is composed by organizing into a square four different samples of the same image filtered with different palettes.

The outcome of this simple example will be a console application that given an input image will produce a second image as described above. This application will use the Thread Model and Aneka for distributing the execution of the steps required to perform the filter.

NOTE: in the state transition diagram it also appear a *Rejected* state. This state is related to the reservation infrastructure that is not active at the moment. In simple terms, WorkUnit instances can be executed with a reservation; this means that a specific slot in the system has been reserved for their execution that is identified by an id. The *Rejected* state come into play when a WorkUnit instance provides to the system a reservation identifier that is not valid or expired. The diagram does not show the *ReScheduled* state that appears when a WorkUnit, while running, is terminated by the infrastructure and put into the scheduling queue for being executed again. This could happen because the execution slot assigned to the instance is expired and cannot be extended.

4.2 *Application Structure*

The source code of this application can be found into the Aneka installation directory under the *Examples/Tutorials/ThreadDemo* directory. There is a convenient Visual Studio 2005 Project that simplifies the build process, but that is not essential for completing the tutorial.

In order to implement the application we will organize the whole application into the three main classes:

- *Aneka.Examples.ThreadDemo.WarholFilter* (see WarholFilter.cs): this class performs the filter of the image and given a picture produces another picture that is the same size of the original and remaps its colors.

- *Aneka.Examples.ThreadDemo.WarholApplication* (see WarholApplication.cs): this class is responfsible for:

    - setting up the AnekaApplication instance;

    - configuring it with for the Thread Model;

    - creating the AnekaThread instances and starting their execution;

    - waiting for the completion of the AnekaThread instances and assembling the four images produced into a single image.

- *Aneka.Examples.ThreadDemo.WarholDriver* (see Program.cs): this class constitutes the main entry point of the application and is in charge of parsing the command line parameters, creating and configuring the WarholApplication class, and starting its execution. In case the command line parameters are not correct the class displays a simple help that explains to ther user how to launch the application.

A summary view of the operations exposed by the three classes can be seen in Figure 4 (next page).

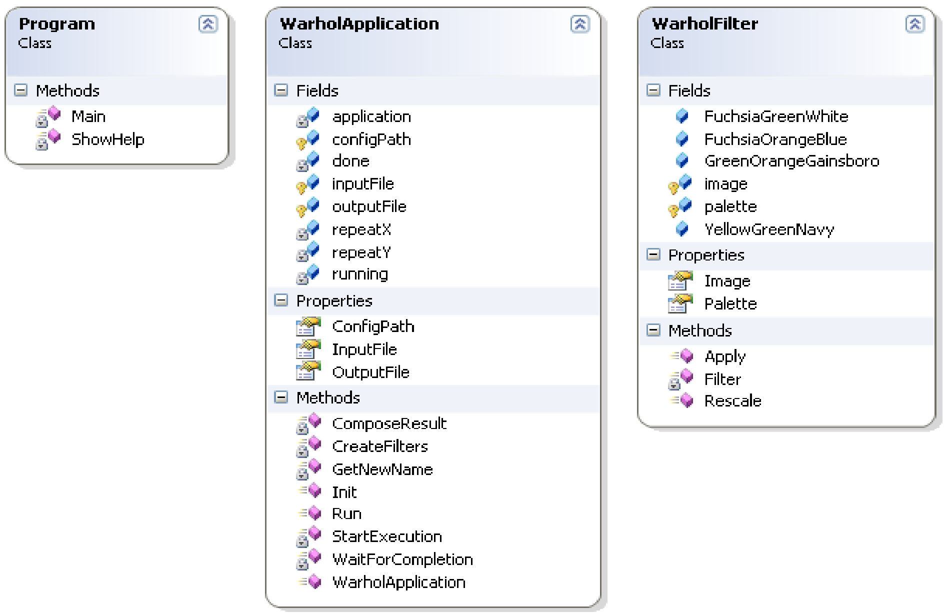


Figure 4 - Class diagram.

These three classes are compiled into an executable (wahrolizer.exe) representing the console application that perform the filter. The application can be started by the command line as follows:

warholizer input_file [output_file] [conf_file]

where:

input_file: path to the image taken as a input for the filter [mandatory]

output_file: path to the file where the filtered image will be saved to [optional]

conf_file: path to the configuration file for connecting to Aneka [optional]

The only mandatory parameter is input_file. If the user does not provide any save path for the output image the application will automatically create a file named *[input-file-name]*.wahrol.*[input-file-ext]* where *[input-file-name]* and *[input-file-ext]* are respectively the name and the extension of the input file. For example:

marilyn.jpg => marilyn.wharol.jpg

If the output file already exists the application will overwrite the file without asking the user permission. For what concerns the settings used to connect to Aneka the user can specify them into an xml file. A sample xml configuration file is already provided (see conf.xml) and it contains the default values that are used when the user does not specify a configuration file. This file whose content is displayed in Figure 4 can be used a starting point for exploring the configuration settings of Aneka and creates customs configurations.

In the next sections we will describe the implementation of the filter, the main steps carried out by the *WarholApplication* to perform the distributed filtering.

```xml
<?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 5 - Aneka Configuration File (conf.xml).

4.3 *WarholFilter: Filter Implementation*

The *WarholFilter* class implements the basic operation of remapping the colors of an image into a three color palette that can be specified by the user.

```
// File: WarholFilter.cs
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;

namespace Aneka.Examples.ThreadDemo
{
    /// <summary>
    /// Class WarholFilter. Performs the color remapping and clustering of an input
    /// image according to a specified palette. This class exposes an Image
property
    /// that is set by the user to the input image to filter and that points to the
    /// processed image after the Apply method has been called.
    /// </summary>
    [Serializable]
    public class WarholFilter
    {
       /// <summary>
       /// Sample palette made by Yellow, DarkGreen, and Navy Color constants.
       /// </summary>
       private static readonly Color[] YellowGreenNavy = new Color[3] { ... };
       /// <summary>
       /// Sample palette made by Fuchsia, Orange, and DarkBlue Color constants.
       /// </summary>
       private static readonly Color[] FuchsiaOrangeBlue = new Color[3] { ... };
       /// <summary>
       /// Sample palette made by Green, Orange, and Gainsboro Color constants.
       /// </summary>
       private static readonly Color[] GreenOrangeGainsboro = new Color[3] { ... };
       /// <summary>
       /// Sample palette made by Fuchsia, DarkOliveGreen, and WhiteSmoke Color
       /// constants.
       /// </summary>
       private static readonly Color[] FuchsiaGreenWhite = new Color[3] { ... };
       /// <summary>
       /// Input/Output bitmap.
       /// </summary>
       protected Bitmap image;
       /// <summary>
       /// Gets, sets the input image on which the filter is applied. This
       /// property stores the filtered bitmap after the Apply() method is
       /// called.
       /// </summary>
       public Bitmap Image
       { get { return this.image; } set { this.image = value; } }
```

```
        /// <summary>
        /// Target color palette.
        /// </summary>
        protected Color[] palette;
        /// <summary>
        /// Gets, sets the palette of colors that will be used to remap the imahge.
        /// </summary>
        public Color[] Palette
        { get { return this.palette; } set { this.palette = value; } }

        /// <summary>
        /// Applies the filter.
        /// </summary>
        public void Apply()
        {
           if (this.image == null)
           {
              throw new ArgumentNullException("Image is null!", "image");
           }
           if (this.palette == null)
           {
              throw new ArgumentNullException("Palette is null!", "palette");
           }
           this.image = this.Filter(this.image, this.palette);
        }
        /// <summary>
        /// Remaps the color values of the source image to the color values
contained
        /// in the given palette by clustering them according to their brightness.
        /// </summary>
        /// <param name="image">source image</param>
        /// <param name="palette">target palette</param>
        /// <returns>filtered bitmap</returns>
        protected Bitmap Filter(Bitmap source, Color[] palette)
        {
            // Step 1. reorder the palette according to the color brightness
            // Step 2. identify the minimum (min) and the maximum (max) brightness
            //         values for the source image and creates (max – min) / length
            //         clusters where the length is the size of the palette.
            // Step 3. invoke Rescale and compute the minimum brightness thresold
            //         color values for each cluster.
            // Step 4. for each pixel of the image evaluates the birghtness and find
            //         find the cluster into which the color will be mapped. Set the
            //         color corresponding to that cluster in the output image.
        }
        /// <summary>
        /// Creates an array of threesold values by recursively dividing the range
        /// identified by max – min and putting the values computed into the given
        /// array.
        /// </summary>
        /// <param name="delta">lenght of the subarray in values that will be filled
        /// during the call of the method. </param>
        /// <param name="start">index of the first element of the subarray</param>
        /// <param name="midPoint">thresold value for the middle element</param>
        /// <param name="min">minimum thresold value</param>
        /// <param name="max">maximum thresold value</param>
        /// <param name="values">target array</param>
        protected void Rescale(int delta, int start,
                          float midPoint, float min, float max, float[] values)
        { ... }
    }
}
```

Listing 3 - Class WarholFilter.

Listing 3 provides a summary view of the class. As it can be noticed by the included namespaces there is nothing in this class that relates to any Aneka library. WarholFilter simply defines the operation that is carried out by the AnekaThread instance when it is excuted on the remote computation node. The relevant members of this class are the following:

- *WarholFilter.Image*: this property is used to stored either the input or the output bitmap of the filter. More precisely when an instance of WarholFilter is created this property is set to the image that will be filtered. After the apply method is called, this property references the filtered image.

- *WarholFilter.Palette*: this property references the palette that will be used to remap the color values of the image. The WarholFilter class exposes four ready t use palettes that contain combination of colors that are used in Andy Warhol's paintings:

    - WarholFilter.YellowGreenNavy
    - WarholFilter.FuchsiaOrangeBlue
    - WahrolFilter.FuchsiaGreenWhite
    - WarholFilter.GreenOrangeGainsboro

These palettes will be used by WarholApplication to create the four images.

- *WarholFilter.Apply()*: this method performs some argument checking on the values of the two properties and invokes *WarholFilter.Filter* to generate the processed image whose color values are rescaled to the values in *Palette*.

It is not the purpose of this tutorial to dig into the details of the implementation of the filter (a complete explanation of the filter is given into the comments in the code) but it is worth estimating its complexity.

4.3.1 Complexity Analysis

The computational complexity of the filter is located into the *WarholFilter.Filter* method and it depends on two parameters: the size of the palette and the image dimension ($O(L_{pal},N_{px})$ where L_{pal} is the size of the palette and N_{px} the number of pixel of the image). The steps involved into the filter as pointed out into the listing are the following:

- Step 1: sorting is made by using bubble sort ($O(L_{pal}^2)$)
- Step 2: the entire image is scanned once ($O(N_{px})$)
- Step 3: the thresold computation is $O(L_{pal})$
- Step 4: the entire image is scanned once and for each pixel the palette array is scanned ($O < O(L_{pal} \times N_{px})$)

The complexity of the filter has then an upper bound of $K_1(O(L_{pal}^2) + O(L_{pal}) + O(N_{px}))$. Since $L_{pal} << N_{px}$ it is possible to identify an upper bound with $K_2O(N_{px})$. This upper bound is a rough estimation of the time required to perform one single filter. The application requires four different samples of the filter. Hence we can easily parallelize this step and reduce the overall computation time.

4.4 *WarholApplication: Distributed Filtering Coordination*

The WarholFilter application is responsible of coordinating the distributed execution of the filters and composing their results together. It uses the services of the Thread Model for parallelizing the computation and the services of *WarholFilter* class for performing the single filtered components of the image. In this section we will describe in detail its behavior.

```
// File: WarholApplication.cs
#region Namespaces
using System;
using System.Collections.Generic;       // Ilist<...> class.
using System.Text;                      // StringBuilder class.
using System.IO;                        // IOException (IO Errors management)
using System.Drawing;                   // Image and Bitmap classes.

using Aneka.Entity;               // Aneka Common APIs for all models
using Aneka.Threading;                  // Aneka Thread Model
using System.Threading;                 // ThreadStart (AnekaThread initialization)
#endregion

namespace Aneka.Examples.ThreadDemo
{
    /// <summary>
    /// Class WarholApplication. This class uses the Thread Model for performing
    /// the distributed filter on a given image and creating the Warhol Effect. It
    /// is responsible of managing the interaction with Aneka and controlling the
    /// execution of the distributed application by managinig the AnekaThread
    /// instances required to perform the filter.
    /// </summary>
    public class WarholApplication
    {
        #region Properties
        /// <summary>Path to the input file.</summary>
        protected string inputPath;
        /// <summary>Gets or sets the path to the input file.</summary>
        public string InputPath
        {
            get { return this.inputPath; }
            set
            {
                if ((value == null) || (value == string.Empty))
                {
                   throw new ArgumentException("InputPath is null or empty",
                                               "InputPath");
                }
                this.inputPath = value;
            }
        }
        /// <summary>Path to the output file.</summary>
        protected string outputPath;
        /// <summary>Gets or sets the path to the output file.</summary>
        public string OutputPath
        {
            get { return this.outputPath; }
            set { this.outputPath = value; }
        }
        /// <summary>Path to the configuration file.</summary>
        protected string configPath;
        /// <summary>Gets or sets the path to the configuration file.</summary>
        public string ConfigPath
        {
            get { return this.configPath; }
            set { this.configPath = value; }
        }
        #endregion

        #region Implementation Fields
        /// <summary>
        /// Configuration settings for the Aneka application.
        /// </summary>
        protected Configuration configuration;
        /// <summary>
        /// Application instance that interfaces the client with Aneka.
        /// </summary>
        protected AnekaApplication<AnekaThread, ThreadManager> application;
        /// <summary>
        /// List containing the currently running AnekaThread instances.
        /// </summary>
        protected IList<AnekaThread> running;
        /// <summary>
        /// List containing the WarholFilter instances that have completed the
        /// execution.
        /// </summary>
        protected IList<WarholFilter> done;
        /// <summary>
        /// Number of columns that will compose the final image.
        /// </summary>
        protected int repeatX;
        /// <summary>
        /// Number of rows that will compose the final image.
        /// </summary>
        protected int repeatY;
        #endregion

        #region Public Methods
        /// <summary>Creates an instance of WarholApplication.</summary>
        public WarholApplication() {}
        /// <summary>Applies the filter.</summary>
        public void Run() { ... }
        #endregion

        #region Helper Methods
        /// <summary>
        /// Reads the configuration and initializes the AnekaApplication instance.
        /// </summary>
        protected void Init() { ... }
        /// <summary>
        /// Starts the execution of the distributed application by creating the
        /// filters wrapping them into AnekaThread instances and starting their
        /// execution.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        protected void StartExecution(Bitmap source) { ... }
        /// <summary>
        /// Waits for the completion of all the threads and if some thread has
        /// failed its execution restarts it.
        /// </summary>
        protected void WaitForCompletion() { ... }
        /// <summary>
        /// Collects the processed images from each filter and composes them
        /// into a single image.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        protected void ComposeResult(Bitmap source) { ... }
        /// <summary>
        /// Creates an array of WarholFilter instances each of them configured
        /// with the same input image and a different palette.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        /// <returns>Array of filters.</returns>
        protected WarholFilter[] CreateFilters(Bitmap source) { ... }
        /// <summary>
        /// Creates a file name by adding the given suffix to the given file name.
        /// </summary>
        /// <param name="name">Source file name.</param>
        /// <param name="suffix">String suffix to append to the name.</param>
        /// <returns>A string containing the new file name.</returns>
        protected string GetNewName(string name, string suffix) { ... }
        #endregion
    }
}
```

Listing 4 - Class WarholApplication.

Listing 4 shows the content of the *WarholApplication.cs* file. This file contains the definition of the *WarholApplication* class. We can inspect the code by first dividing the content of the file in five sections delimited by the #region ... #endregion preprocessor directives:

- Namespaces: this region includes all the required namespaces for developing the application. Among the namespaces included three are of a particular interest:
    - *Aneka.Entity (Aneka.dll)*: This namespace contains the definition of the base APIs common to all the models supported by Aneka. Inside this namespace we can find the *AnekaApplication* class, the *Configuration* class, the *WorkUnit* class, and all those types that are part of the core object model. This namespace needs always to be include while developing applications for Aneka.
    - *Aneka.Threading (Aneka.Threading.dll)*: This namespace contains the definition of the APIs for the Thread Model. In particular it contains the definition of the *AnekaThread* and the *ThreadManager* class. This namespace needs to be included when developing applications based on the Thread Model.

      NOTE: The whole set of APIs of a model is generally organized into three namespaces: *Aneka.[model]*, *Aneka.[model].Scheduling*, and *Aneka.[model].Execution*. These namespaces correspond to three different assemblies named with the same convention. When developing applications with a specific model it is generally required to include only the first namespace, since the other two namespaces contain the definition of server side components.

    - *System.Threading (System.dll)*: This namespace contains .NET Threading APIs. When developing applications with the Thread Model it is necessary to initialize AnekaThread instances with a *ThreadStart* object whose definition is contained in the *System.Threading* namespace.

The other namespaces that have been included into the file contains the definition of support classes that have been used to program the WarholApplication class. For example, since the class deals with images it is necessary to include the *System.Drawing (System.Drawing.dll)* namespace.

- Properties: this region contains the three properties that constitute all the parameters required to perform the filter. The path to the input image (WarholApplication.InputPath), the path where to save the output image (*WarholApplication.OutputPath*), and to the path configuration file (*WarholApplication.ConfigPath*). The only mandatory parameter is the path to the input image that must point to an existing file. The other two parameters are configured with the default values if set to null.

- Implementation Fields: this region contains the declaration of the protected members that are used to manage the application. There are five fields declared:

    - Configuration configuration;
    - AnekaApplication<AnekaThread, ThreadManager> application;
    - IList<AnekaThread> running;
    - IList<WarholFilter> done;
    - int repeatX;
    - int repeatY;

  The first four fields constitute infrastructure that is generally required while developing applications based on the Thread Model. As for any other model it is necessary to create a *AnekaApplication* instance that represents the local view of the distributed application. Since *AnekaApplication* is a generic type it needs to be specialized by using the components that identify the model we use. In this case we will use a *AnekaApplication<AnekaThread, ThreadManager>* instance, that specializes the behaviour of *AnekaApplication* for the *Thread Model*. The *AnekaApplication* class is initialized with a set of configuration parameters that are stored in the *Configuration* class. This class allows to customize the behavior of *AnekaApplication* instances. Moreover, two additional data structures are required: a list containing the *running AnekaThread* instances and *done* list containing the filter instances that have completed their execution. It can be noticed that running is a list of *AnekaThread* instances while done is a list of *WarholFilter* instances. The reason for this is because while the thread is running we need to keep a reference to it in order to join the thread and checks its state. Once a thread has completed its execution there is no more need to store a reference to it but we simply keep the *WarholFilter* instance connected to the thread that is accessible by the *AnekaThread.Target* property.

Two more fields have been defined: *repeatX* and *repeatY*. These fields are specific to the application we are developing and maintain the information about the number of columns and rows that compose the final image.

- Public Methods: the public methods of the class are its default constructor and the *Run* method that starts the execution of the filter. The content of this method will be analyzed in detail in the next section.

- Helper Methods: in order to make the more understandable the code, the body of the Run method has been divided into logical steps that have been encapsulated into helper methods. In particular we can identify four major steps:

  - *WarholApplication.Init*: initializes the *AnekaApplication* instance with the selected *Configuration* instance.

  - *WarholApplication.StartExecution*: initializes the *running* and *done* lists, creates the *AnekaThread* instances, and starts their execution.

  - *WarholApplication.WaitForCompletion*: waits for the completion of all the *AnekaThread* instances in the *running* list and eventually restarts their execution if they failed. When this method returns, the running list is empty and the done list contains the reference to the *WarholFilter* instances created.

  - *WarholApplication.ComposeResult*: iterates on the done list and compose the final output image by arranging the filtered bitmaps into an image with *repeatX* columns and *repeatX* rows. This method saves the output bitmap into the *WarholApplication.OutputPath* if set, otherwise it generates a new name as described in Section 4.2.

  This region contains two more methods that are invoked by the previous one that are in charge of creating the list of different filters that will be applied to the image (*WarholApplication.CreateFilters*) and of generating a new name for the output file (*WarholApplication.GetNewName*).

In the following we will explore in more details the single steps of the application.

```
namespace Aneka.Examples.ThreadDemo
{
    ...
    public class WarholApplication
    {
        ....
        /// <summary>
        /// Application instance that interfaces the client with Aneka.
        /// </summary>
        protected AnekaApplication<AnekaThread, ThreadManager> application;
       ....
```

```
        /// <summary>
        /// Applies the filter.
        /// </summary>
        public void Run()
        {
            if (File.Exists(this.inputPath) == false)
            {
                throw new FileNotFoundException("InputPath does not exist.",
                                             "InputPath");
            }
            try
            {
                // Initializes the AnekaApplication instance.
                this.Init();

                // read the bitmap
                Bitmap source = new Bitmap(this.inputPath);

                // create one filter for each of the four slices that will
                // compose the final image and starts their execution on
                // Aneka by wrapping them into AnekaThread instances...
                this.StartExecution(source);

                // wait for all threads to complete...
                this.WaitForCompletion();

                // collect the processed images and compose them
                // into one single image.
                this.ComposeResult(source);

            }
            finally
            {
                // we ensure that the application closes properly
                // before leaving the method...
                if (this.application != null)
                {
                    if (this.application.Finished == false)
                    {
                        this.application.StopExecution();
                    }
                }
            }
        }
    }
}
```

Listing 5 - Run() method.

Listing 5 reports the content of the *Run* method. As we can notice body of the method sequentially calls the logical steps identified before and ensures that the application instance is closed if some error occurs. In the finally block we simply check that application instance is not null and whether it has completed its execution by looking at the *AnekaApplication<W,M>.Finished* property. If the application is not finished, the *AnekaApplication<W,M>.StopExecution()* method is invoked to terminate its execution.

NOTE: The lines of code contained in the finally block identify a common programming pattern for all the models supported by Aneka. This pattern ensures the resources allocated by the AnekaApplication instance are properly released and no work unit is left running on Aneka.

We can now explore the single steps and see how to set up the AnekaApplication instance and configure it for its execution.

```
namespace Aneka.Examples.ThreadDemo
{
    ...
    public class WarholApplication
    {
        ....

        /// <summary>
        /// Application instance that interfaces the client with Aneka.
        /// </summary>
        protected AnekaApplication<AnekaThread, ThreadManager> application;
        ....

        /// <summary>
        /// Reads the configuration and initializes the AnekaApplication
        /// instance.
        /// </summary>
        protected void Init()
        {
            Configuration configuration = null;
            if (string.IsNullOrEmpty(this.configPath) == true)
            {
               this.configuration = Configuration.GetConfiguration();
            }
            else
            {
               this.configuration =
Configuration.GetConfiguration(this.configPath);
            }

            // we set this force to false because
            // we want to handle the resubmission
            // of failed threads.
            this.configuration.SingleSubmission = false;

            // we initialize the AnekaApplication instance with the
            // selected configuration object and the components required
            // for the Thread Model
            this.application =
            new AnekaApplication<AnekaThread,
ThreadManager>(this.configuration);

    }
}
```

Listing 6 - Init() method.

Listing 6 reports the body of the *Init* method. The method performs to very basic steps:

- Configuration setup: if the configuration path has been the static method *Configuration.GetConfiguration(string)* is invoked to read to read the information form the given file. If the configuration path has not been set, the static method *Configuration.GetConfiguration()* is called. This method will first look for the default configuration file (in this case: warholizer.exe.config) and if not found it will create a default configuration object. The default values for the *Configuration* class can been seen in the *conf.xml* file (see Figure 5). Once the configuration instance has been created the value of *SingleSubmission* is set to false. The reason for this is because in case some threads are failed they will be restarted again.

NOTE: The Thread Model is naturally based on the coordinated execution of remotely executable threads that can be started at any time. The *SingleSubmission* property forces the behavior of the *AnekaApplication* to a single submission of all the *AnekaThread* instances that have been explicitly added to the *AnekaApplication* before calling the method *AnekaApplication.SubmitExecution*. Since with the Thread Model the *AnekaThread* instances are not explicitly added to the AnekaApplication setting SingleSubmission to true would lead to an unexpected behaviour and it could cause the premature termination of the application. This problem is even worse when we want to handle the resubmission of the failed threads. It is then a general recommendation to **not to use SingleSubmission set to true when using the Thread Model**.

- AnekaApplication initialization: this step is accomplished by simply initializing the *application* field and passing as parameter the Configuration object that has been obtained at the previous step. Since the *AnekaApplication* class is a generic type its initialization implies specifying the actual types used by the application. In the case of the Thread Model we will specialize the *AnekaApplication* class by using *AnekaThread* in place of the *WorkUnit* and *ThreadManager* in place of *IApplicationManager*.

The next step of the process is performed by the *WarholApplication.StartExecution* method that takes as input the Bitmap instance read from the input file and creates the *AnekaThread* instances as described in Listing 7.

```
namespace Aneka.Examples.ThreadDemo
{
    ...
    public class WarholApplication
    {
        ....
        /// <summary>
        /// Application instance that interfaces the client with Aneka.
        /// </summary>
        protected AnekaApplication<AnekaThread, ThreadManager> application;
          ....

        /// <summary>
        /// Starts the execution of the distributed application by creating the
        /// filters wrapping them into AnekaThread instances and starting their
        /// execution.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        protected void StartExecution(Bitmap source)
        {
            this.running = new List<AnekaThread>();
            WarholFilter[] filters = this.CreateFilters(source);

            // creates an AnekaThread for each filter
            foreach (WarholFilter filter in filters)
            {
                AnekaThread thread = new AnekaThread(new
                                   ThreadStart(filter.Apply), this.application);
                thread.Start();
                this.running.Add(thread);
            }
        }
    }
}
```

Listing 7 - StartExecution(Bitmap) method.

The interesting bits in this method are concentrated within the foreach loop. For each *WarholFilter* instance that has been created a new instance of *AnekaThread* is initialized and configured to run the *WarholFilter.Apply* method. Each *AnekaThread* also need to have a reference to the *AnekaApplication* it belongs to. The second statement simply starts the execution of the *AnekaThread* instance by calling *AnekaThread.Start()*.

These two statements constitute the common operations required to configure and start a *AnekaThread* instance. We also add this instance to the list of running threads in order to keep track of its reference and being able to get the results once the AnekaThread has completed its execution.

```
namespace Aneka.Examples.ThreadDemo
{
    ...
    public class WarholApplication
    {
        ....

        /// <summary>
        /// Starts the execution of the distributed application by creating the
        /// filters wrapping them into AnekaThread instances and starting their
        /// execution.
        /// </summary>
        protected void WaitForCompletion()
        {
            this.done = new List<WarholFilter>();
            bool bSomeToGo = true;
            while (bSomeToGo == true)
            {
                foreach (AnekaThread thread in this.running)
                {
                    thread.Join();
                }
                for (int i = 0; i < this.running.Count; i++)
                {
                    AnekaThread thread = this.running[i];
                    if (thread.State == WorkUnitState.Completed)
                    {
                        this.running.RemoveAt(i);
                        i--;
                        WarholFilter filter = (WarholFilter) thread.Target;
                        this.done.Add(filter);
                    }
                    else
                    {
                        // it must be failed...
                        thread.Start();
                    }

                }
                bSomeToGo = this.running.Count > 0;
            }
        }
    }
}
```

Listing 8 - WaitForCompletion() method.

This method exposes another classic synchronization pattern that is used while creating multi-threaded applications: threads synchronization. In this specific case we simply want to implement a barrier for all threads. This can be easily done by calling the *AnekaThread.Join()* method on all the instances that we have started and that are contained in the *running* list.

The method uses a while loop that will terminate once all the results have been collected. Inside the while loop two basic steps are performed:

- Wait for thread completion: the method invoke the *AnekaThread.Join()* method on all the threads contained in the *running* list. This call makes the application to wait until the thread terminate.

- Check thread results: once all the threads have terminated we iterate again the *running* list to check whether some thread has failed its execution or not by looking at the *State* property. If the thread has successfully completed its execution its state is set to *WorkUnitState.Completed*. In this case we simply remove the thread from the *running* list and add *WarholFilter* instance referenced by the *AnekaThread.Target* property into the *done* list. If the other cases the thread is simply restarted.

The loop terminates when the *running* list is empty. This means that all the threads have successfully completed their execution and all the filters have been collected into the *done* list.

```
namespace Aneka.Examples.ThreadDemo
{
    public class WarholApplication
    {
        ....
        /// <summary>
        /// Starts the execution of the distributed application by creating the
        /// filters wrapping them into AnekaThread instances and starting their
        /// execution.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        protected void ComposeResult(Bitmap source)
        {
            Bitmap output = new Bitmap(source.Width * this.repeatX,
                                     source.Height * this.repeatY,
                                          source.PixelFormat);

            Graphics graphics = Graphics.FromImage(output);
            int row = 0, col = 0;
            foreach (WarholFilter filter in this.done)
            {
                graphics.DrawImage(filter.Image, row * source.Width,
                                                 col * source.Height);
                row++;
                if (row == this.repeatX)
                {
                    row = 0;
                    col++;
                }
            }
            graphics.Dispose();
            if (string.IsNullOrEmpty(this.outputPath) == true)
            {

                this.outputPath = this.GetNewName(this.inputPath, "warhol");
            }
            output.Save(this.outputPath);
        }
}
```

Listing 9 - ComposeResult(Bitmap) method.

The next logical step is constituted by the processing of the result and the creation of the final output image. These tasks are accomplished into the *AnekaThread.ComposeResult* method whose content is reported in Listing 9. The interesting bits in this method are concentrated within the foreach loop. For each *WarholFilter* instance that is contained into the *done* list the processed image exposed by the *WarholFilter.Image* property is drawn into the final output image in the position identified by the *row* and *col* local variables. The member field *repeatX* is used to identify the end of a line and move to the next colum.

After the output bitmap has been composed it is saved to the file pointed by *OutputPath* property or to an automatically generated file name by invoking the *WarholApplication.GetNewName* method.

```
namespace Aneka.Examples.ThreadDemo
{
    ...
    public class WarholApplication
    {
        ....
        /// <summary>
        /// Creates an array of WarholFilter instances each of them configured
        /// with the same input image and a different palette.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        /// <returns>Array of filters.</returns>
        protected virtual WarholFilter[] CreateFilters(Bitmap source)
        {
            WarholFilter[] filters = new WarholFilter[4];

            WarholFilter one = new WarholFilter();
            one.Image = source;
            one.Palette = WarholFilter.FuchsiaGreenWhite;
            filters[0] = one;

            WarholFilter two = new WarholFilter();
            two.Image = source;
            two.Palette = WarholFilter.YellowGreenNavy;
            filters[1] = two;

            WarholFilter three = new WarholFilter();
            three.Image = source;
            three.Palette = WarholFilter.FuchsiaOrangeBlue;
            filters[2] = three;

            WarholFilter four = new WarholFilter();
            four.Image = source;
            four.Palette = WarholFilter.GreenOrangeGainsboro;
            filters[3] = four;

            this.repeatX = 2;
            this.repeatY = 2;

            return filters;
        }
    }
}
```

Listing 10 - CreateFilters(Bitmap) method.

The last method that we want to explore is the WarholApplication.CreateFilters method that is responsible of creating all the filter instances and define the number of rows and columns into which the final image will be organized. By separating the creation of filters into this method we can easily customize the output image by simply overriding this method and, for example, creating a final image that is composed by 9 samples of the original images or simply changing the colors of the palette.

4.5 *Program: Putting all together*

The program class implements a simple command line parser that reads the arguments of given by the user, checks whether they are correct, configures the *WarholApplication* class and starts the execution of the filter by invoking the *Run()* method. If the user has not provided the right parameters a simple command line help is show.

The class defines only two static methods: one is the entry point of the application (*Program.Main(string[])*) and the other one shows the command line help (*Program.ShowHelp()*).

```
using Aneka;
```

```
// File: Program.cs
namespace Aneka.Examples.ThreadDemo
{
    /// <summary>
    /// Class Program. Virtualizes the execution of WarholFilter by using the
    /// Thread Model. This class simply parses the command line arguments
passed
    /// to the process and sets up the WarholApplication.
    /// </summary>
    public class Program
    {
        /// <summary>
        /// Creates an array of WarholFilter instances each of them configured
        /// with the same input image and a different palette.
        /// </summary>
        /// <param name="args">Command line arguments.</param>
        static void Main(string[] args)
        {
            try
            {
                 Logger.Start();

                 if (args.Length >= 2)
                 {
                     string inputFile = args[0];
                     string outputFile = args[1];
                     string confFile = null;
                     if (File.Exists(inputFile) == false)
                     {
                         Console.WriteLine("warholizer: [ERROR] input file" +
                                           "[{0}] not found. EXIT.",
inputFile);
                         return;
                     }
                     else
                     {
                         // the infput file exists...
                         // now we check for the configuration file.
                         if (args.Length == 3)
                         {
                             confFile = args[2];
                             if (File.Exists(confFile) == false)
                             {
                                 Console.WriteLine("warholizer: [ERROR] " +
                                         "configuration file [{0}] not found.

EXIT",
                                         inputFile);
                                 return;
                             }
                         }
                         // now we check for the out file to simply issue
                         // a warning if the file exists...
                         if (File.Exists(outputFile) == true)
                         {
                             Console.WriteLine("warholizer: [WARNING] output
file"
                             + " [{0}] already exists and it will be
overwritten.",
                             inputFile);
                         }
                    }
                    // ok at this point we have the following conditions
                    // 1. inputPath exists
                    // 2. confFile exists
                    // we can start the application..
                    WarholApplication app = new WarholApplication();
                    app.InputPath = inputFile;
                    app.OutputPath = outputFile;
                    app.ConfigPath = confFile;
                    try
                    {
                        app.Run();
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine("warholizer: [ERROR] exception:");
                        Console.WriteLine("\tMessage: " + ex.Message);
                        Console.WriteLine("\tStacktrace: " + ex.StackTrace);
                        Console.WriteLine("EXIT");

                        IOUtil.DumpErrorReport(ex, "Aneka Thread Demo - Error
Log");
                    }
                 }
                 else
                 {
                     Program.ShowHelp();
                 }
             }
             finally
             {
                 Logger.Stop();
             }
        }
        /// <summary>
        /// Shows a command line help about the usage of the application.
        /// </summary>
        static void ShowHelp() { .... }
    }
}
```

Listing 11 - Program class.

It is important to notice that **each application** using the Aneka APIs requires a basic *try { .... } catch { ... } finally { ... }* block that is used to ensure a proper initialization of the environment as well as a proper release of resources. In particular the it is necessary to perform the following steps:

- Initialize the *Logger* class at the beginning of the *try {...}* block. This operation activates the logger and all the resources required to provide a solid and reliable logging. This operation is generally not required because the logger will automatically initialize at the first call but it is a good practice to explicitly call the logger.

- Provide a catch block intercepting all the exceptions occuring in the main thread. It is possible to use the *IOUtil.DumpErrorReport(....)* method to properly log the content of the exception to a file. The method provides different overloads that allow users to specify for example an header or the name of the log file. The version used in the example creates a log file named ***error.YYYY-MM-DD_HH-mm-ss.log*** to the application base directory.

- Finalize the logger in the *finally {...}* block by calling *Logger.Stop()*.This operation ensures that all the resources required by the logging are properly released and that there are no threads still running at the end of the application.

NOTE: among all the three operations listed above the most important one is the finalization of the logger. The Logger is a static class that provides logging capabilities to all the components of Aneka and uses an asynchronous model to log messages and raise log events. If the user forgets to call Logger.Stop() the thread used to raise log events will keep running thus preventing the termination of the application. It is a safe practice to put this operation within a finally block so that it is ensured that it is executed.

4.6 *Compiling and building the Application*

4.6.1 Building the demo in Visual Studio 2005

It is possible to build and run the application by simply opening the Visual Studio 2005 Project *ThreadDemo.csproj* in the *ThreadDemo* directory and build the project. Visual Studio will created the executable warholizer.exe along with all the libraries required to run in the *ThreadDemo\bin\Debug* directory (Configuration: Debug).

Visual Studio 2005 will also copy the *conf.xml* and the *marilyn.jpg* file into the *bin\Debug* directory of the application. These two files can be used to test the execution of the application.

4.6.2 Building the demo from the command line

If you do not have the Visual Studio 2005 installed but you have c# 2.0 compiler (let us assume that the compiler is the one shipped with .NET framework SDK and that is called *cs.exe*) it is possible to compile the application from the command line.

Dependencies

The first step that is required is identifying the dependencies that this application relies on to execute:

1. Most of the support classes that we have used to build the application are defined in the *System.dll* assembly that is referenced by default and contained in the Global Assembly Cache (GAC).

2. To perform the operations on the images we have used the Bitmap class which is defined in the System.Drawing namespace (GAC: *System.Drawing.dll*).

3. In order to use the Thread Model we used the types defined in the Aneka.Threading namespace that is implemented in the *Aneka.Threading.dll*.

4. Any application that uses the Aneka APIs has an implicit dependency on the following assemblies:

    a. Aneka.dll (Namespaces: Aneka, Aneka.Entity, Aneka.Security)

    b. Aneka.Data.dll (Namespaces: Aneka.Data, Aneka.Data.Entity)

    c. *Aneka.Util.dll* (Namespace: Aneka (utility classes))

The complete set of dependencies is then given by: *System.dll*, *System.Drawing.dll*, *Aneka.dll*, *Aneka.Threading.dll*, *Aneka.Data.dll*, *Aneka.Util.dll*. We can find the libraries that relate to Aneka into the *[Aneka Installation Directory]\bin* directory. The easiest thing to do is then copy these libraries to the ThreadDemo directory. The other two libraries are registered in the GAC, hence we do not need to copy them.

Compilation

Once we have copied the required libraries into the ThreadDemo directory we can invoke the C# 2.0 compiler to compile the three files (*WarholFilter.cs*, *WarholApplication.cs*, and *Program.cs*) that compose the application with the following command line:

```
csc /r:System.dll /r:System.Drawing.dll /r:Aneka.dll
/r:Aneka.Data.dll /r:Aneka.Util.dll /r:Aneka.Threading.dll
/t:exe /out:warholizer.exe Program.cs WarholFilter.cs
WarholApplication.cs
```

The compilation process will create the *warholizer.exe* executable in the *ThreadDemo* directory.

4.6.3 Running the application

In order to run the application it is necessary to connect to have Aneka installed either on the local machine or on a remote machine that can be reached through a TCP connection. We assume, for the sake of simplicity, that Aneka is running on the local machine with the default installation (port: 9090). In this case we can simply run test the application by running the from the command line the following:

```
warholizer.exe marilyn.jpg
```

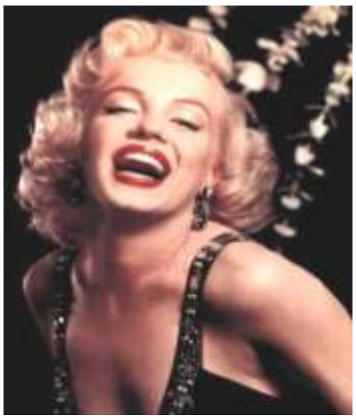
This application will produce the file *marilyn_warhol.jpg* in the same directory. We can also provide a different name (for example *foo.jpg*) of the output file by executing the follwing:

```
warholizer.exe marilyn.jpg foo.jpg
```

If we need to customize the way in which the application connects to Aneka. We can simply edit the configuration file *conf.xml* (for example we need to change the address where the application should connect) and run the following:

```
warholizer.exe marilyn.jpg foo.jpg conf.xml
```

Figure 6 shows the input file and a possible outcome of the execution of warholizer on the given input file.



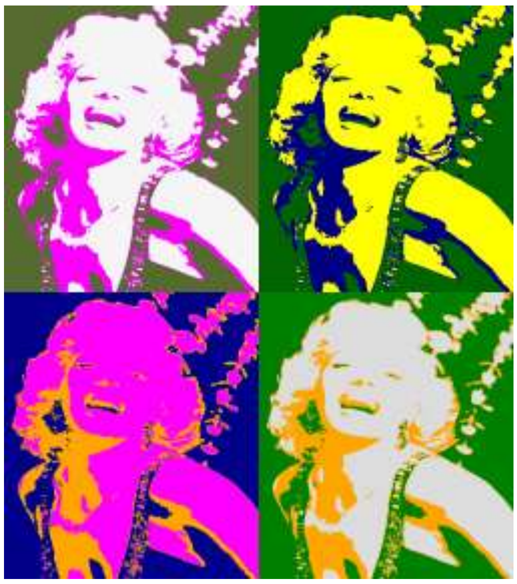


Figure 6 - Input (left) and output (right) images (not in original sizes).

5 File Management

The Aneka application model introduced in Section 2 also allows the management of files that might be of support for its execution or an outcome of the execution of threads. Specifically files can be:

- Common files required by all the tasks of the application.
- Input files for specific threads.
- Output files of specific threads.

As it can be noticed there is no concept of output file for the entire application.

In order to show how to take advantage of the file management features implemented in Aneka we will modify the current example and we will introduce explicit file management. In particular we will demonstrate how to add files that are of use for the entire application and how to manage input and output files for single thread instances.

5.1 *Aneka File APIs*

Aneka encapsulates the required information about a file into the *Aneka.Data.Entity.FileData* class that is defined in the *Aneka.Data.dll*. This class is used to represent all the different kind of files that are managed by Aneka.

Listing 12 provides an overall view of the properties of interest for the *FileData* class and the related enumeration. In the listing, only the properties that are important from an user point of view have been reported.

```
namespace Aneka.Data.Entity
{
    /// <summary>
    /// Enum FileDataAttributes. Describes the different set of attributes that a
    /// a FileData instance can have. Attributes are used to convey additional
    /// information about the file.
    /// </summary>
    [Flags]
    public enum FileDataAttributes
    {
       /// <summary>
       /// No Attributes.
       /// </summary>
       None          =   0,
       /// <summary>
       /// The file is located into the file system of the Aneka application and
       /// not stored in a remote FTP server.
       /// </summary>
       Local =   1,
       /// <summary>
       /// The file is transient and does not represent neither an input file nor
       /// a final output file. At the moment this attribute is not used.
```

```
        /// </summary>
        Transient    =   2,
        /// <summary>
        /// The file is optional. This attribute only makes sense for output files
        /// and notifies Aneka that the file might or might not be produced by a
        /// task as outcome of its execution.
        /// </summary>
        Optional     =   4
    }
    /// <summary>
    /// Enum FileDataAttributes. Describes the different set of attributes that a
    /// a FileData instance can have. Attributes are used to convey additional
    /// information about the file.
    /// </summary>
    [Flags]
    public enum FileDataType
    {
        /// <summary>
        /// No type, not used.
        /// </summary>
        None         =   0,
        /// <summary>
        /// The file is an input file to a specific WorkUnit.
        /// </summary>
        Input =   1,
        /// <summary>
        /// The file is an output file to a specific WorkUnit.
        /// </summary>
        Output=   2,
        /// <summary>
        /// The file is a common input file to the application and it will be
        /// available as input to all the WorkUnit.
        /// </summary>
        Shared=   4,
        /// <summary>
        /// This is just the sum of all the values of the enumeration.
        /// </summary>
        All    =   Input + Output + Shared
    }
    /// <summary>
    /// Class FileData. Represents a generic file in the Aneka application model.
    /// Within Aneka files are automatically moved to execution nodes and collected
    /// back to the client manager once the WorkUnit instances are completed
    /// successfully. The properties exposed by this class allow Aneka to automate
    /// file management and make it transparent to the user.
    /// </summary>
    [Serializable]
    public class FileData
    {
            /// <summary>
            /// Gets or sets the full path to the file.
            /// </summary>
```

```
          public string Path { get { ... } set { ... } }
          /// <summary>
          /// Gets or sets the path that the file will have to the remote execution
          /// node. This property is mostly of concern for Input and Output files
          /// since shared files are generally placed where Aneka requires them.
          /// </summary>
          public string VirtualPath { get { ... } set { ... } }
          /// <summary>
          /// Gets or sets the name of the file. The name is kept separate from the
          /// path because instances of FileData can be used in a cross-platform
          /// environment where the interpretation of the path differs from node to
          /// node.
          /// </summary>
          public string FileName { get { ... } set { ... } }
          /// <summary>
          /// Gets or sets a string representing the unique identifier of the owner
          /// of the file. The owner can be a specific WorkUnit or the Application
          /// instance. In the first case the file is an Input or an Output file,
          /// in the second case it is a Shared file.
          /// </summary>
          public string OwnerId { get { ... } set { ... } }
          /// <summary>
          /// Gets a boolean value indicating whether the file is local to the file
          /// file system of the application or located in a remote storage
          /// facility.
          /// </summary>
          public bool IsLocal { get { ... } }
          /// <summary>
          /// Gets or sets the type of the file.
          /// </summary>
          public FileDataType Type { get { ... } set { ... } }
          /// <summary>
          /// Gets or sets the collection of attributes that are attached to the
          /// FileData instance.
          /// </summary>
          public FileDataAttributes Attributes { get { ... } set { ... } }

          ......

    }
}
```

Listing 12 - FileDataAttributes, FileDataType, and FileData.

A file in Aneka is identified by the following combination of values: *OwnerId*, *Type*, *Path*/*VirtualPath*. An *OwnerId* identifies the owner of the file, which represents the entity (*WorkUnit* or *Application*) that requires or produces the file. In most of the cases, the user will not be asked to provide this information that is automatically added while adding files to the *WorkUnit* instances or to the collection of the shared files of the application. Of major importance are the *Type* and *Path*/*VirtualPath* propeties.

5.1.1 File Types

The Type property is used to express the nature of the file, which determines how Aneka manages. Three types are used by the user:

- *FileDataType.Shared*: this is an input file to the entire application and its absence prevents the execution of the application itself. Shared files are meant to be available to all the *WorkUnit* instances that the application is composed of.

- *FileDataType.Input*: this is an input file to a specific *WorkUnit* and its absence prevents that specific *WorkUnit* to be executed, causing its failure.

- *FileDataType.Output*: this is an output file of a specific *WorkUnit*. If the file it is not produced as outcome of the execution of the *WorkUnit* and it is not marked as optional the corresponding *WorkUnit* is considered failed.

Shared and Input files are automatically moved from their origin (the local file system to the application or a remote file server) to the execution node of the *WorkUnit* and the user does not need to pre-upload files. Output files are moved from the execution node to the local file system or a remote file server once the *WorkUnit* has completed its execution or aborted. In this second case Aneka will look at the expected output files and will collect only those that have been generated.

5.1.2 Path vs VirtualPath, and FileName

The management of files is completed by the information given through the *Path*, *VirtualPath*, and *FileName* properties. These three values help the framework to locate the file during all the life-cycle of the application. At first we can distinguish two major contexts: the execution node and the external world. In the execution context the *FileData* instance is identified and located by looking at the *VirtualPath* property, whereas in the external world (should this be the user local file system or a remote file server) the *Path* property is used to locate the file.

The reason why there are two distinct properties is because this allows the user to change the name of files and provides an higher degree of flexibility in managing files. Moreover, there could be some legacy applications that produce file in a specific path and whose behavior cannot be changed; when these applications are executed within Aneka the framework should still be able to properly collect the files. The value of the virtual path is automatically inferred is left unspecified by the user.

In addition, a specific property has been designed for keeping the file name. The reason for this, is because Aneka has been designed to run on an cross-platform environment in which the interpretation of the path information is not uniform and might lead to not properly locating files. By keeping the name separate from the path the framework will always be able to collect the file.

5.1.3 File Attributes

Aneka allows to attach additional information to the *FileData* instance to simplify the management of files and provide advanced features such as pulling and pushing files from remote FTP servers or by means of other kind of protocols.

The Attributes property contains the collection of attributes attached to the file, which are defined by the *FileDataAttributes* enumeration. Among all the available attributes there are only two, which are of interest for the user:

- *FileDataAttributes.IsLocal*: this attribute is set by default when creating a FileData instance and it identifies the corresponding file as belonging to the local file system of the application. Local input files are pushed into the Aneka storage facility from the client computer, while local output files are downloaded into the client computer once the *WorkUnit* that produced them has completed. If a file is not local it resides on a remote storage and Aneka will pull input files from the remote storage, and push output files to the remote storage.

NOTE: In case of remote file it is important to provide the Aneka runtime will all the information necessary to pull the files into the Aneka Storage facility. Such information can be saved into the Aneka configuration file under the property group "StorageBuckets". A storage bucket is a collection of properties in the form of name-value pairs that are helpful to connect to a remote storage server and upload/download a file. In the case of an FTP storage the user name and password are be required. Each storage bucket is identified by a name that is used by the FileData instance to map a remote file with the required credentials to access the remote storage: the FileData.StorageBucketId property will store the name of the corresponding storage bucket for remote files.

- *FileDataAttributes.Optional*: this attribute is mostly related to output files and identifies files that might (or might not) be produced as outcome of the execution of a *WorkUnit*. By setting this attribute, a *WorkUnit* is not considered failed is some (or all) of these files are not present in the remote execution node.

Other attributes are internally used are not of interest from a user point of view.

5.2 *Providing File Support for Aneka Applications.*

The *AnekaApplication* class and the *WorkUnit* class provide users with facilities for attaching files that are required by the application. This can be done either by creating *FileData* instances or by simply providing the file name and the type. The use of *FileData* instances is more appropriate in cases where it is necessary to differentiate the path from the virtual path, or whether we need to map a remote file.

5.2.1 Adding Shared Files

Listing 13 shows how to add shared files to the application by either providing only the file name or a *FileData* instance. It is possible to specify a full path for the file, in case no path is given the file will be searched in the current directory. It is important to notice that in case the user decide to provide a *FileData* instance the value of the *OwnerId* and the Type property will be overridden with the values shown the listing, which are the application unique identifier and the *FileDataType.Shared* type respectively.

```
        ....
        /// <summary>
        /// Reads the configuration and initializes the AnekaApplication instance.
        /// </summary>
        protected void Init()
        {
            if (string.IsNullOrEmpty(this.configPath) == true)
            {
                this.configuration = Configuration.GetConfiguration(();
            }
```

```
            else
            {
                this.configuration =
Configuration.GetConfiguration(this.configPath);
            }

            // we set this force to false because
            // we want to handle the resubmission
            // of failed threads.
            this.configuration.SingleSubmission = false;

            // we turn off the ShareOutputDirectory property
            // we will explain this later..
            this.configuration.ShareOutputDirectory = false;

            // we initialize the AnekaApplication instance with the
            // selected configuration object and the components required
            // for the Thread Model
            this.application =
           new AnekaApplication<AnekaThread, ThreadManager>(this.configuration);

           // we add the application input file as a shared file
```

```
            // because we need to have the file available for all
            // the threads..
            this.application.AddSharedFile(this.inputPath);

              // NOTE:
              // alternatively we can explicitly create a FileData instance as
              // follows and add it
              // FileData shared = new FileData(this.application.Id, this.inputPath,
             //                                 FileDataType.Shared);
             // this.application.AddSharedFile(shared);

    }
    ....
```

Listing 13 - Adding shared files to an Aneka application.

In this example, we have provided the application with the input image that will be processed by all the threads. This file will be available on the remote execution node of each of the threads composing the application.

5.2.2 Adding Input and Output Files

In order to leverage files for image processing instead of Image instances we need to change the interface of the WarholFilter class by replacing the Image property with two string properties representing the input file required by the filter and the output file name generated as a result of the processing. Listing 14 shows the changes that have to be applied to the *WarholFilter* class.

```
        // /// <summary>
        // /// Input/Output bitmap.
        // /// </summary>
        // protected Bitmap image;
        // /// <summary>
        // /// Gets, sets the input image on which the filter is applied. This
        // /// property stores the filtered bitmap after the Apply() method is
        // /// called.
        // /// </summary>
        // public Bitmap Image
        // { get { return this.image; } set { this.image = value; } }

        /// <summary>
        /// input file.
        /// </summary>
        protected string input;
        // /// <summary>
        // /// Gets, sets the input file name of the source image for the filter.
        // /// </summary>
        public string InputFile
        { get { return this.input; } set { this.input = value; } }
        /// <summary>
        /// output file.
        /// </summary>
        protected string output;
        // /// <summary>
        // /// Gets, sets the output file name of the result image.
        // /// </summary>
        public string OutputFile
        { get { return this.output; } set { this.output = value; } }
```

Listing 14 - Adding shared files to an Aneka application.

Listing 15 shows how to add an output file to a thread instance. As reported, the process of adding an input file is exactly the same. The *AddFile* method provides a different overload allowing to specify the *FileData* instance as parameter. It is important to remember that the value of the *OwnerId* that needs to be passed is the identifier of the thread instance. In any case the method will override the property value so that it matches the identifier of the thread instance. Moreover, the *WorkUnit* class also exposes the *InputFiles* and *OutputFiles* collections (*IList<FileData>*) where the user can directly add *FileData* instances.

```
        /// <summary>
        /// Starts the execution of the distributed application by creating the
        /// filters wrapping them into AnekaThread instances and starting their
        /// execution.
        /// </summary>
        /// <param name="source">Source bitmap.</param>
        protected void StartExecution(Bitmap source)
        {
            this.running = new List<AnekaThread>();
            WarholFilter[] filters = this.CreateFilters(source);
           // initialize a counter to create output names.
           int i = 0;
            // creates an AnekaThread for each filter
            foreach (WarholFilter filter in filters)
            {
                AnekaThread thread = new AnekaThread(new ThreadStart(filter.Apply),
                                                 this.application);

                 // we set the input and the output file names so that the filter
                // instance can know which file to read and to write. For both
                // files we only need the name of the file because they will be
                // the current execution directory.
                 filter.InputFile = Path.GetName(this.inputPath);
                 filter.OutputFile = this.GetNewName(filter.InputFile,
i.ToString());

                // we add the output file to the thread, in this case we do not need
                // any input file because the image to process is a shared file and
                // it becomes an input file by default.
                // In order to add an input file we can change the value of the
                // FileDataType enumeration to Input.
                thread.AddFile(filter.OutputFile, FileDataType.Output,
                               FileDataAttributes.Local);
               thread.Start();
                this.running.Add(thread);

                 // we increment the counter so that we can dreate different output
                // file names (one for each thread instance)
                i++;
            }
        }
```

Listing 15 - Adding input and output files to tasks.

5.2.3 Using Files on the Remote Execution Node

In the previous steps we have modified the source code of the example for providing the application and the thread instances with input files and collecting output files. Both shared and input files will be located in the current execution directory of the thread instance that will also be the destination path of the output file. We will now change the code of the *WarholFilter.Apply* method in order to read and write files instead of using the in-memory representation of the image.

```
        /// <summary>
        /// Applies the filter.
        /// </summary>
        public void Apply()
        {
           // NOTE: this is the old code leveraging the in memory representation
           //       of the image.
           //
           // if (this.image == null)
           // {
           //    throw new ArgumentNullException("Image is null!", "image");
           // }
           if (this.palette == null)
           {
              throw new ArgumentNullException("Palette is null!", "palette");
           }

           Bitmap source = new Bitmap(this.input);
           Bitmap result = this.Filter(source, this.palette);
           result.Save(this.output);
```

```
            // NOTE: this is the old code leveraging the in memory representation
            //       of the image.
            //
            // this.image = this.Filter(this.image, this.palette);
        }
```

Listing 16 - Reading and writing input and output files on the remote node.

As it can be noticed from the listing, there is no specific operation that has to be performed inside the method to access the input and shared files or to write output files. In this case we simply need to create an in memory representation of the input image and once the filtered image is created save with the selected file name for the output file.

5.2.4 Collecting Local Output Files

Once the thread has completed it is possible to access the content of local output file from the file system local to the user. The value of the *Path* property and the configuration settings of the Aneka application will determine the location of the output file.

If the *Path* property contains a rooted path, this will be location where the file will be found. Otherwise, the file will be stored under the application working directory that is represented by *Workspace* property of the *Configuration* class. In this directory, a subdirectory whose name is represented by the *Home* property of the application instance will be created to store all the output files. Since it might be possible that all the output files produced by different threads could have the same name, Aneka allows to store the content of each *WorkUnit* instance in a separate directory that is named after the *WorkUnit.Name* property. This value is automatically filled when the user creates an AnekaTask instance and is in the form “Thread-N” where N is a sequential number. By default Aneka saves the output of each thread instance in a separate directory, in order to turn off this feature it is sufficient to set the value of *Configuration.ShareOuput-Directory* to false.

In the current example, we have given a different name to each output file. Hence, there is no need to save the results in a separate directory and we can set the value of *Configuration.ShareOutputDirectory* to false.

Finally, if the value of *Configuration.Workspace* is not set, the default working directory is taken as reference directory.

```
          ....
          /// <summary>
          /// Starts the execution of the distributed application by creating the
          /// filters wrapping them into AnekaThread instances and starting their
          /// execution.
          /// </summary>
          /// <param name="source">Source bitmap.</param>
          protected void ComposeResult(Bitmap source)
          {
              Bitmap output = new Bitmap(source.Width * this.repeatX,
                                    source.Height * this.repeatY,
                                         source.PixelFormat);

              Graphics graphics = Graphics.FromImage(output);
              int row = 0, col = 0;

              // we collect the value of the output directory of the application.
              string outputDir = Path.Combine(this.configuration.Workspace,
                                         this.application.Home);

              foreach (WarholFilter filter in this.done)
              {
                  // NOTE: old code, we now read the processed image from the
                  //       output files attached to the filter.
                  //
                  // graphics.DrawImage(filter.Image, row * source.Width,
                  //                                  col * source.Height);

                  string outputFile = Path.Combine(outputDir, filter.OutputFile);
                  Bitmap image = new Bitmap(outputPath);
                  graphics.DrawImage(image, row * source.Width, col * source.Height);

                  row++;
                  if (row == this.repeatX)
                  {
                      row = 0;
                      col++;
                  }
              }
              graphics.Dispose();
              if (string.IsNullOrEmpty(this.outputPath) == true)
              {

                  this.outputPath = this.GetNewName(this.inputPath, "warhol");
              }
              output.Save(this.outputPath);
          }
           ...
```

Listing 17 - Accessing output files on the user local file system.

Listing 17 shows how to collect the information about the location of the output file in the user local file system. As previously mentioned the location of the local output files is influenced mostly by the configuration settings. In the example discussed being all the output files different in names we can instruct Aneka to store them in the same directory.

The changes applied to the *ComposeResult* method simply locate the output files and read the corresponding *Bitmap* instance in order to compose and save the final image.

5.3 Observations

This section has provided an overview of the support for file management in Aneka. Differently from other distributed computing middlewares, Aneka automates the movement of files to and from the user local file system or remote storage servers to the Aneka internal storage facility. The major reason behind this design decision is simplify as much as possible the user experience and to automate those task that do not really require the user intervention.

The basic features of file management in Aneka have been demonstrated by modifying the discussed example in order to support shared, input and output files. The example includes only files that belong or will be saved into the file system local to the user, while the use of files located in remote servers has not been demonstrated. The use of this feature requires a proper configuration of the Aneka Storage service, which goes beyond the scope of this tutorial.

6 Aneka Thread Model Samples

The *examples* directory in the Aneka distribution contains some ready to run applications that show how is it possible to use the services provided by Aneka to build non-trivial applications. The examples concerning the Thread Execution Model are the following:

- *Mandelbrot*
- *ThreadDemo* (within the Tutorials folder)

The *ThreadDemo* has been fully explored in this tutorial. For what concerns the Mandelbrot example we will simply give some hints on how to explore the Visual Studio Projects related to the sample and see how the Thread Model has been used to distributed the application.

6.1 Mandelbrot

6.1.1 Mandelbrot Set

The *Mandelbrot set* is a set of complex numbers for which the following iteration:

$$z = z_0$$

$$z_{n+1} = z_n^2 + z$$

does not diverge to infinity. This means that there exist a number N that can be considered the upper bound of the previous iteration. What makes interesting the Mandelbrot set is the fact that when applied to complex numbers it generates a bi-dimensional figure whose border does not simplify if magnified. In other word Mandelbrot set creates very interesting and fascinating fractals. These fractals can be easily generated by a computer program by using the following algorithm:

```
Escape Time Algorithm
```

```
1. let be W and H the size of the image we want to compute
2. let be z = 0+i0
3. then the range of complex values is [0+i0, W+iH]
4. for each complex number c = x+iy in [0+i0, W+iH]
    4.1 set s = true
    4.2 for i:0 to max_iterations do:
     4.2.1 compute z = z*z + c
          4.2.2 if |z| > 2 then set s = false and break
    4.3 if s is true c is in the Mandelbrot set (color: black)
    4.4 if s is false c is not in the Mandelbrot set (color: gradient)
5. end
```

Listing 18 - Mandelbrot set drawing.

The Escape Time algorithm is based on the assumption that no complex number with a modulus bigger than 2 can be part of the Mandelbrot set. This can be used as a quick condition to check whether the sequence of numbers generated for each c diverges or not. For those complex numbers that belong to the Mandelbrot set this condition will always hold and the iterations will continue indefinitely. The algorithm then imposes a maximum number of iterations after which the given number can be reasonably considered part of the Mandelbrot set. The algorithm presented does not guarantee a perfect drawing of the Mandelbrot set but the bigger it is the number of iterations the more precise is the resulting Mandelbrot set.

6.1.2 Parallel Mandelbrot computation

As it can be noticed from Listing 12, the computation that is performed for each complex number in the range (that is for each pixel of the image) is identical and there is no relation between the values computed for one complex number and next one evaluated. This makes the determination of the Mandelbrot set an *embarrassingly parallel* problem.

It is possible to define a distributed version of the previous algorithm that:

1. Divides the range [0+i0, W+iH] into N rectangles whose size is wxh.

2. Computes in parallel the Mandelbrot set for each rectangle [w*k+ih*j, w*(k+1) + h*(j+1)] where k = W / w, and j = H / h.

3. Composes the result into a single image.

Moreover, it is possible to provide a general offset to the image and a zoom factor that allow us to explore the self similarity of the border of the Mandelbrot set.

6.1.3 Mandelbrot Sample

Figure 7 shows the Mandelbrot sample, which is available in the Aneka distribution. This sample uses the Thread Model for parallelizing the computation of the Mandelbrot set as described in the previous paragraph.

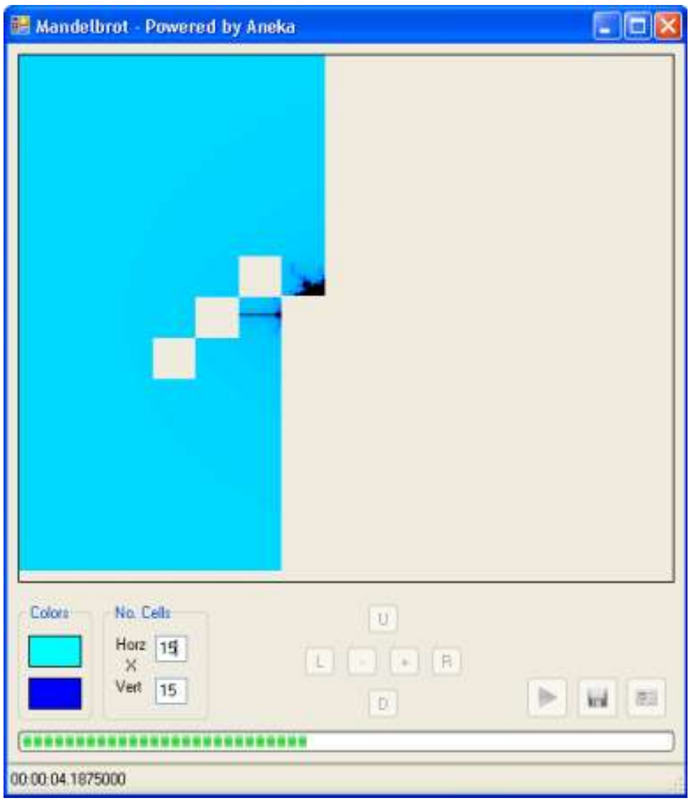


Figure 7 - Distributed Mandelbrot

The application is accessible from the Start Menu -> Programs -> Manjrasoft -> Aneka -> Examples -> Mandelbrot and the related files are located under the [Aneka Installation Directory]\examples\Mandelbrot directory.

The sample is constituted by two components:

- The user interface application (Mandel.exe) that displays the Mandelbrot set and allows to distribute its computation on Aneka.

- The MandelThread.dll library containing the definition of two classes:

  - MandelThread: performs the computation of the Mandelbrot set on a given rectangular region of the image.

  - Complex: support class that represents a complex number.

The user interface creates as many AnekaThread instances as the number of cells that have been specified by the user and starts their execution on Aneka. As soon as a AnekaThread instance completes it updates the image displayed.

6.1.4 Conclusion

In this section we only have discussed the issues concerning the parallelization of a Mandelbrot set computation by using the Thread Model. The whole application is more complex and goes beyond the scope of this tutorial. It concerns also user interface management in a multi-threaded environment.

7 Conclusions

In this tutorial we have introduced the Thread Model for developing distributed applications based on remotely executable threads with Aneka. The Thread Model allows developers to quickly virtualize multi-threaded applications with Aneka. It introduces the concept of *AnekaThread* that represents a thread that is executed on a remote computing node in the Aneka network. The *AnekaThread* class exposes a subset of the operations offered by the *System.Threading.Thread* class, this makes the transition from a local multi-threaded application to a distributed multi-threaded application straightforward.

As happens for local threads a *AnekaThread* is configured with a *ThreadStart* object that wraps the information required run a method. It is possible to start, join, and abort a thread in the same manner as we do with local threads. Few restrictions apply to the execution of remote threads. As explained in the tutorial, *AnekaThread* instances cannot be paused or run static methods and they do not support all the asynchronous operations of the *Thread* class. Given these limitations, the Thread Model remains still appealing for developers that want to take advantage of distributed computing systems without learning a new programming model.

In order to explain and illustrate the approach to the development of distributed application by using the Thread Model a simple application has been developed: warholizer. This application is a multi-threaded image filters that reproduces the Warhol Effect on a given picture by leveraging the computation on Aneka. In particular the following aspects have been discussed:

- What is the Thread Model and how it relates with the common .NET threading APIs.
- How to create and configure an *AnekaApplication* for the Thread Model.
- How to create and configure an *AnekaThread.*
- How to control the execution of a Thread Model application by using *AnekaThread* instances.
- How to implement the common synchronization patterns used in multi-threaded applications.
- How to structure the source code of an application that is based on the Thread Model.
- How to compile and build a working example from the command line.
- How to manage shared, input, and output files for the Thread Model.

This tutorial has also introduced the general notions concerning a distributed system based on Aneka and the essential information for using the client APIs that are common to all programming models. For a more complete and detailed description of the behavior of these APIs it is possible to explore the APIs documentation.