MANJRASOFT PTY LTD

Task Submission Web Service

Aneka 5.0

Manjrasoft

This documentation explains how to use the Task Submission Web Service for the management and monitoring of applications based on the Task Programming Model on top of Aneka. The task web service provides developers with a language independent interface for submitting bag of tasks application from any client that is capable of interacting with a Web Service. It describes the interaction flow for the creation of an application, the submission of tasks, and the monitoring of their execution. It also underlines the major differences that are to be taken into account with respect to the usual in-process client API described in "Developing Task Model Applications" and gives some guidance on the deployment of the web service.

Table of Contents

1 Prerequisites 1
2 Introduction 1
3 Interaction Flow 2
4 Operations 3
4.1 User Operations 5
4.1.1 AuthenticateUser 5
4.2 Application Operations 5
4.2.1 CreateApplication 5
4.2.2 QueryApplication 5
4.2.3 QueryApplicationStatus 5
4.2.4 AbortApplication 6
4.3 Job Operations 6
4.3.1 SubmitJobs 6
4.3.2 QueryJob 6
4.3.3 QueryJobStatus 6
4.3.4 AbortJob 6
4.4 More Information 6
5 Inside the Job 7
5.1 Task Item Definition 7
5.2 File Management 10
6 Storage Service Configuration 13
7 Advanced Topics and Technical Details 14
7.1 Disconnected Mode vs. Connected Mode 14
7.1.1 Task Persistence in the Scheduling Service 16
7.1.2 Task Persistence in the Execution Service 17
7.1.3 Observations 18
8 Web Service Configuration and Deployment 18
8.1 Installation on Windows Operating Systems 19
8.2 Installation on Unix-based Systems with Mono 19
9 Conclusions 19

Copyright © 2010 Manjrasoft Pty Ltd.

1 Prerequisites

The intended audience of this documentation are software developers that want to leverage the capabilities of the *Task Programming* Model implemented in Aneka for the execution of *Bag of Tasks* applications. In order to fully understand the concepts exposed in this document the user should be familiar with the architecture of Aneka and the basics of the *Task Programming Model*. Also a basic understanding of Web Services and their operation mode is required.

2 Introduction

The current release of Aneka (Aneka 2.0.2.0) contains a Web Service project allowing the submission of simple tasks to Aneka. Task submission leverages the *Task Programming Model*, which is used to execute tasks on the Aneka Cloud. The Web Service is fully compliant with the *Aneka Distributed Application Model* and allows the submission of jobs, which are mapped to task instances, within the context of an application.

The general reference scenario is depicted in Figure 1.

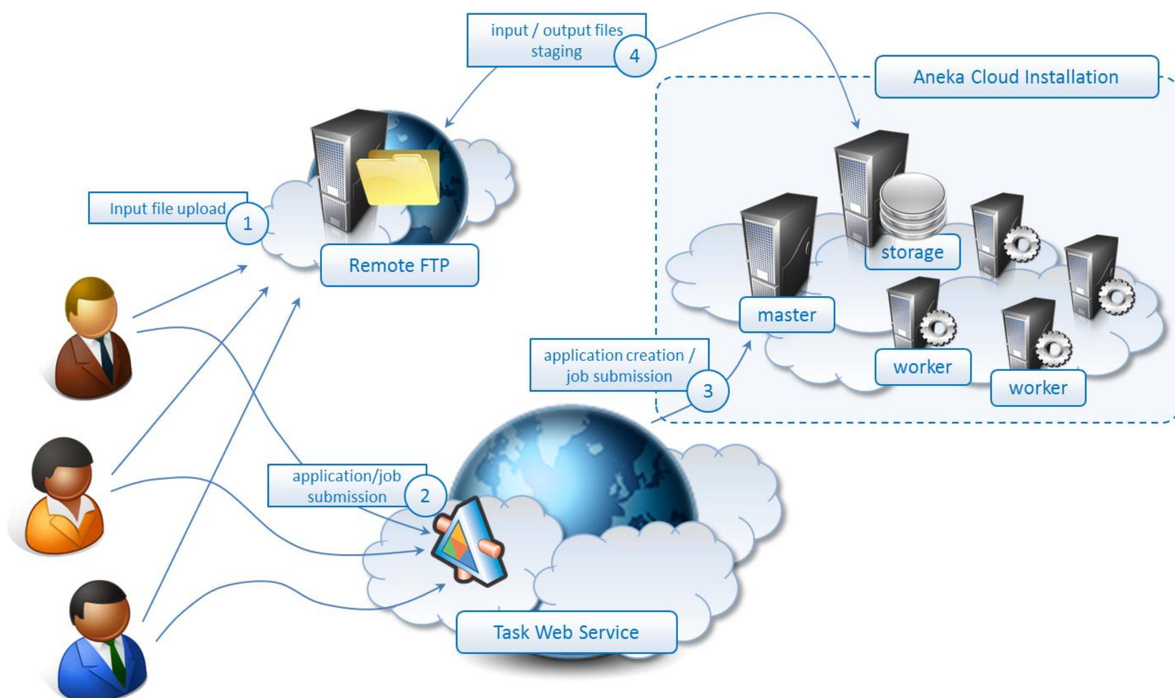

Figure 1 - Accessing Aneka through the Web Service.

In the given scenario a collection of users interacts with the web service to submit tasks to Aneka. The web service supports the upload of *shared files* for the application, *input files* for the jobs, and allows specifying *output files* that will all be staged in/out from or to a remote storage server, most likely FTP. The common operation sequence is the following:

Copyright © 2010 Manjrasoft Pty Ltd.

1. Users upload input and shared files to a remote storage server accessible through the network **from the Aneka installation**.
2. They submit the application (and then the jobs) through the web service.
3. The web service will instruct the Aneka cloud to create a corresponding application instance and schedule the jobs.
4. The *Aneka Storage Service* will pull in all the required files for the applications.
5. The *Aneka Task Scheduling Service* will execute the jobs and result files will be stored in the storage service.
6. The Aneka Storage Service will push the output files to remote storage server.
7. By interacting with the web service users can monitor the application and terminated it if necessary.

The current implementation supports FTP as remote storage server and also S3 given that the *Aneka Storage Service* is properly configured in order to access an S3 storage (See: *Aneka.Data.S3.dll*). The rest of the interaction for what concerns application management is completely controlled by the Web Service interface.

3 Interaction Flow

The figure below describes the sequence of operations that are required to create and manage a distributed application, and then submit, query (and eventually abort) jobs.

- The first operation to accomplish once the web service client is connected is to authenticate the client by providing a simple username and password credentials. Once this operation is completed successfully the web service will return an authentication token that can be used for the other operations.
- As happens for the in-process client APIs it is necessary to create an application before submitting tasks to Aneka. This is done by providing the web service with general details of the application such as the display name, the information about the shared files used by the application, and the credentials to authenticate the request against Aneka. If the operation is successful the web service returns a string with an unique identifier of the application and no error, otherwise useful information about the exception occurred in case of error are reported.
- Once the application has been created it goes into the *"Running"* state and the users can query the state of the application and get all the information about it as shown in the diagram. At the same time Aneka will automatically pull inside the Storage Service all the shared files that are required by the application to execute. At the end of this process it is possible to submit jobs and obtain the corresponding unique identifies that can be used to query their state. Again, if errors occur the web service return information about the exception occurred. If there is a null identifier in the list returned by web service the corresponding job has not been successfully submitted.
- If the job has been submitted successfully it is possible to query its status and get information about its execution. It is either possible to obtain the full information about the job or simply its status. If the job successfully completes and it has output files, these files will be automatically staged out to the remote storage server indicated in the job submission details by the "Aneka Storage Service" upon job completion.

Copyright © 2011Manjrasoft Pty Ltd.

- For both jobs and applications it is possible to abort their execution at any time. The abortion of a job simply terminates the execution of the job if it was already running or prevents it from being executed if it was queued. The abortion of the application implies the abortion of all the jobs that are not terminated at the time of the abortion request.

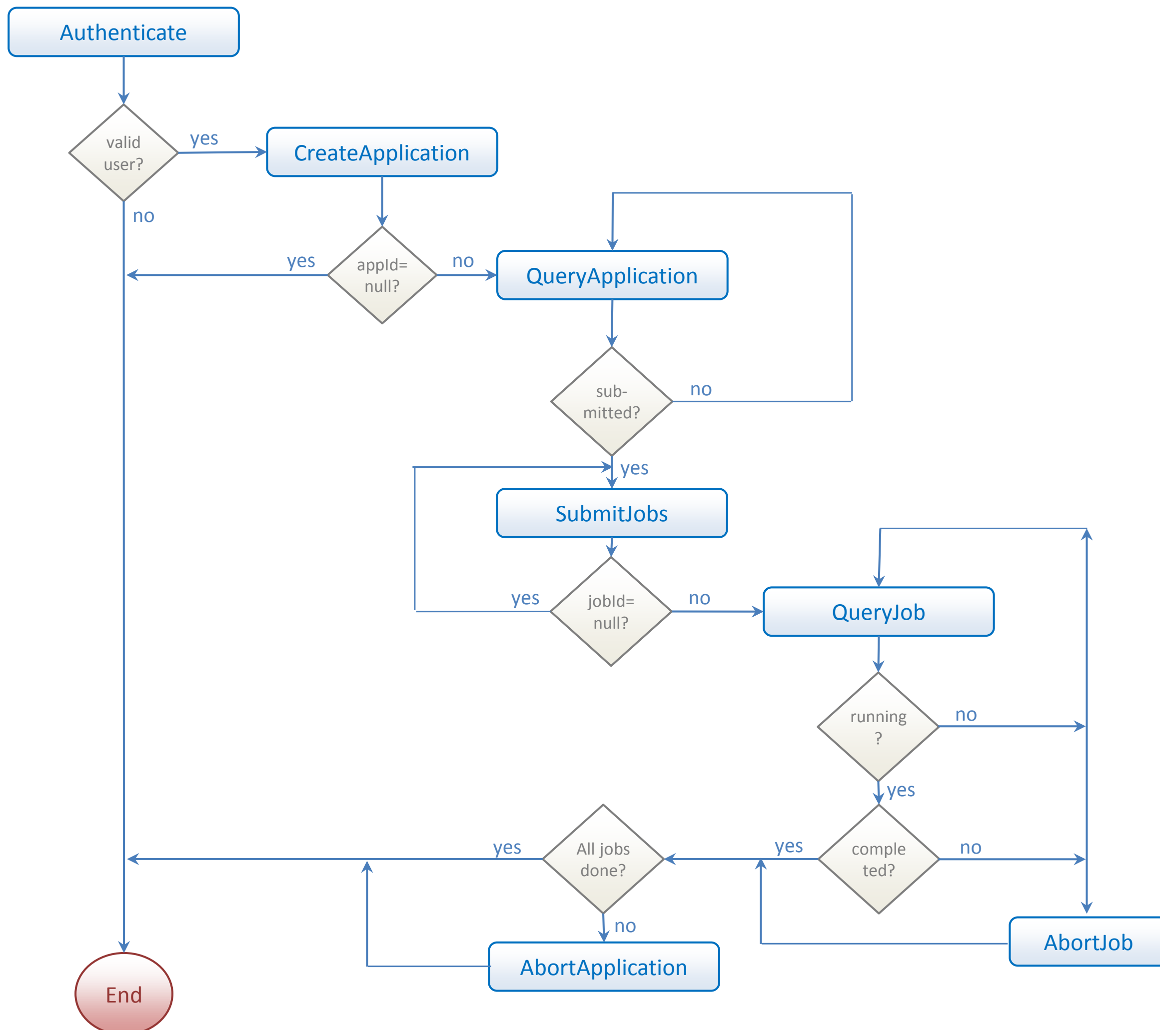

Figure 2 - Web Service Interaction Flow.

4 Operations

A complete and more detailed diagram of the Web Service interface and of the types of the parameters required by the methods is depicted in the diagram in Figure 3.

The diagram is organized into three major class groups:

- **Web service class**. This is the implementation of the Web Service (only the public interface).
- **Input parameter classes**. These classes model all the information that is given to the web service as input parameters. The root class of this group is the Request class, which exposes only one property: *UserCredential* an array of bytes that

Copyright © 2011Manjrasoft Pty Ltd.

contains the binary form of the authentication token used to authenticate the request against the Aneka Cloud. All the inherited classes can be divided into two groups: application-wise (*ApplicationRequest*, *ApplicationQueryRequest*, and *ApplicationAbortRequest*) and job-wise (*JobSubmissionRequest*, *JobQueryInfo*, and *JobAbortRequest*).

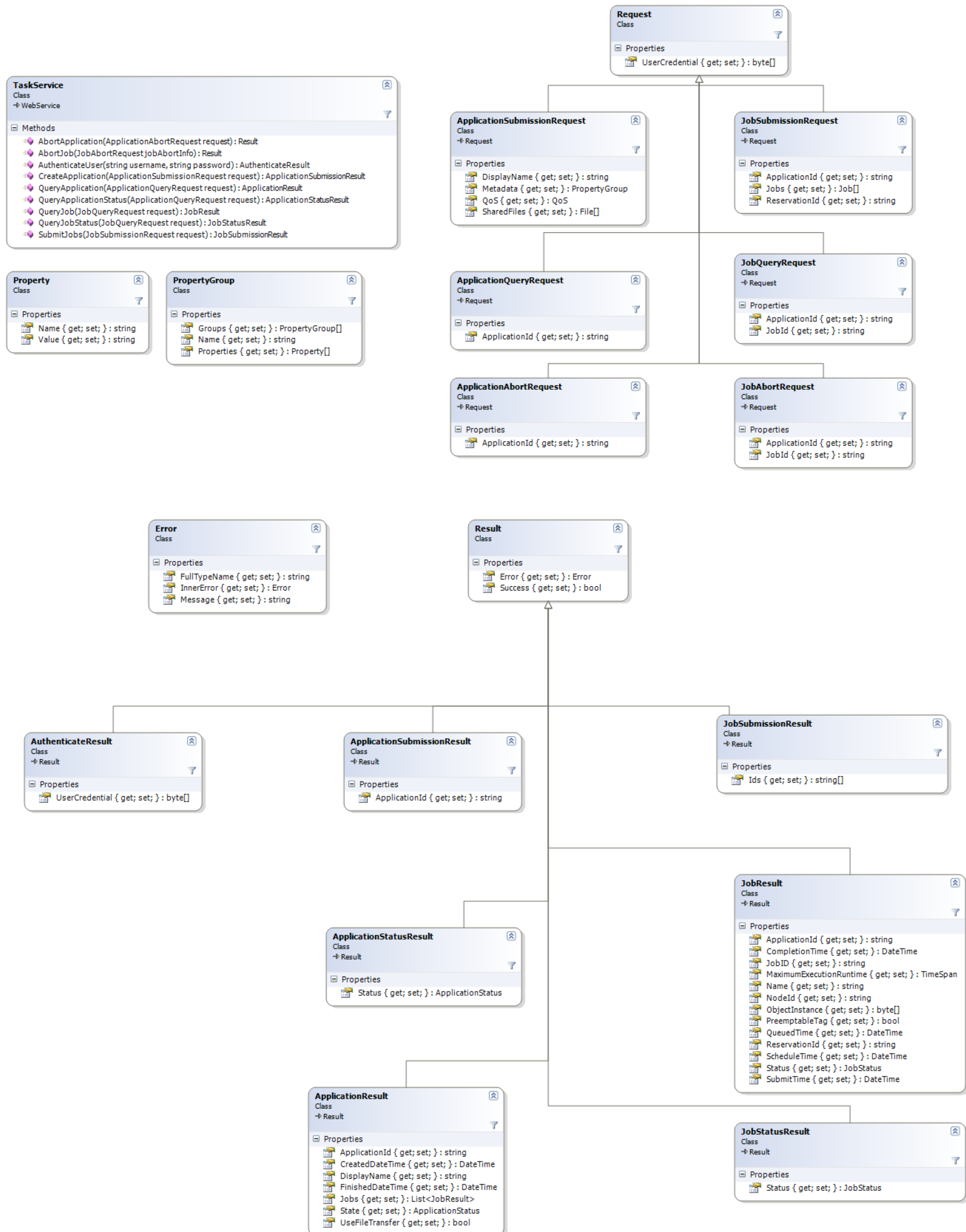

Figure 3 - Task Service Object Model.

Copyright © 2011Manjrasoft Pty Ltd.

- **Output parameter classes**. These classes model all the information that is returned by the web service as result of a method call. The root class of this group is the *Result* class, which exposes two properties: *Success* and *Error*. The first property is boolean and provides information on whether the operation has been performed successfully and the latter is of type *Error* and provides information on the exception occurred during the execution of the web service method if there is any failure. This class maintains simple information about the error message and full type name of the exception. All the inherited classes can be divided into two groups: application-wise (*ApplicationSubmissionResult*, *ApplicationResult*, and *ApplicationStatusResult*) and group-wise (*JobSubmissionResult*, *JobResult*, and *JobStatusResult*).

A more detailed of all the available operations exposed by the Web Service follows.

4.1 User Operations

4.1.1 AuthenticateUser

It authenticates the user against the Aneka Cloud and returns the user credential token that can be used to authenticate all the other requests. The object returned by this method is of type *AuthenticateResult* which contains the binary token for the authentication. If the authentication has not been successful information about the error occurred are provided through the *Error* property and *Success* is set to false.

4.2 Application Operations

4.2.1 CreateApplication

It creates a new application for the submission of jobs on the Aneka Cloud. This operation requires a parameter of type *ApplicationSubmissionRequest* which contains the information for the creation of the application such as the display (or friendly name) of the application and the user credential under which the application is created. This operation returns an instance of *ApplicationSubmissionResult* containing the unique id of the application, if successful the Success property is set to true otherwise additional information is provided through the *Error* property.

4.2.2 QueryApplication

It Queries the status of the application. This method requires one instance of *ApplicationQueryRequest* containing the information about the specific application. It returns an *ApplicationResult* instance that contains the information about the status of the application with information on how many tasks have been submitted and their status. If the operation is unsuccessful the value of *Success* is set to false and more details are given with the *Error* property.

4.2.3 QueryApplicationStatus

It queries the status of the application (only). This method requires one instance of *ApplicationQueryRequest* containing the information about the specific application. It returns an *ApplicationStatusResult* instance that contains the information about the status

Copyright © 2011Manjrasoft Pty Ltd.

of the application. If the operation is unsuccessful the value of *Success* is set to false and more details are given with the *Error* property.

4.2.4 AbortApplication

It terminates the execution of an application (and all its dependent jobs). This method takes a string parameter containing the application identifier and a second parameter of type *ApplicationAbortRequest* containing the application unique identifier and the user credentials. It returns a Result instance whose *Success* property indicates the outcome of the operation. If the operation fails major details are given in the *Error* property.

4.3 Job Operations

4.3.1 SubmitJobs

It submits a collection of Jobs. This method requires a *JobSubmissionRequest* instance containing the collection of jobs to submit, the identifier of the application the jobs belong to, and the user credentials. It returns an instance of *JobSubmissionResult* containing the list of the corresponding job identifiers of the successfully submitted jobs. If - for some reason - there are jobs that failed submission, the corresponding job identifier is set to null, the value of *Success* is set to false and major details are given in the *Error* property.

4.3.2 QueryJob

It queries the status of a job. This method requires one instance of *JobQueryRequest* containing all the information required for identifying the job together with the user credentials. It returns a *JobResult* instance that contains the information about the status of the job, other metadata, and the binary data of its instance if available. If the operation is unsuccessful the value of *Success* is set to false and more details are given with the *Error* property.

4.3.3 QueryJobStatus

It queries the status of a job (only). This method requires one instance of *JobQueryRequest* containing all the information required for identifying the job together with the user credentials. It returns a *JobStatusResult* instance that contains the information about the status of the job. If the operation is unsuccessful the value of *Success* is set to false and more details are given with the *Error* property.

4.3.4 AbortJob

It terminates the execution of a job. This method takes an instance of *JobAbortRequest* containing the details of the job to abort together with the user credentials. It returns a *Result* whose property *Success* informs about the outcome of the operation. If unsuccessful this property is set to false and major details are given in the *Error* property.

4.4 More Information

By downloading the WSDL of the web service it is possible to obtain all the information required for submitting jobs to Aneka. In order to obtain the WSDL of the web service it is

Copyright © 2011Manjrasoft Pty Ltd.

necessary to append the *?WSDL* query string to the address of the web service as reported in the following example:

```
http://localhost/Aneka.2.0/TaskService.asmx?WSDL
```

5 Inside the Job

In Aneka an application is the logical wrapper that coordinates together the execution of different units. This is particularly true in the case of the *Task Programming Model* where the entire work is carried out by tasks that are submitted by the user. Therefore, the abstraction of task becomes fundamental and constitutes basic building block of distributed applications for this model. In the object model exposed by the *Task Web Service* a *task* is mapped to a *job*, which is further decomposed in a collection of *task items*, and an application becomes a collection of jobs.

Figure 4 shows all the classes that define the object model expressing a job. The fundamental class is represented by Job which exposes the following properties:

- **ReservationId**: it allows user to assign the job to a specific reservation that is represented by the identifier. If the Job does not have any value set for this property, it will be automatically assigned to the value of *JobSubmissionInfo.ReservationId*.
- **InputFiles**: represents the collection of input files that the job is expected to find at the remote execution node for executing. These files have to be stored in some file server that can be accessed by Aneka to pull them into the system when needed. Files are expressed by means of the *File* class.
- ***OutputFiles***: represents the collection of output files that job produces as a result of its execution. These files will then be uploaded to a remote storage server once the job is completed. Files are expressed by means of the *File* class.
- **Tasks**: represents the collection of tasks that compose the job. These are mapped to the common tasks available in each *Parameter Sweep* based model and another special task (*NativeTaskItem*), which allows the submission of tasks that are native to a specific runtime.

The job instance submitted to the web service is wrapped around an *AnekaTask* instance and the *UserTask* property is set to a corresponding *Aneka.Tasks.BaseTask.CompositeTask* that will executed in sequence the task defined in the *Job.Tasks* array.

5.1 Task Item Definition

The operational logic of a job is defined by composing together a collection of task items. The job object model provides developers with a set of predefined tasks that can be easily composed together in order to perform the most common operations that are performed in a batch environment. These predefined tasks have a correspondence with the tasks defined in the *Aneka.Tasks.BaseTasks*. These tasks are:

- **CopyTaskItem**: this object is mapped to a *CopyTask* instance. The *CopyTask* class allows users to copy a file. At the time of writing the usefulness of this task is quite limited since it does not allows remote file transfer. The reason for this limitation

Copyright © 2011Manjrasoft Pty Ltd.

is because Aneka still does not support the remote file transfer from a generic server (http, windows share, ftp) to an executor node. This feature will be soon (well, not that soon) implemented.

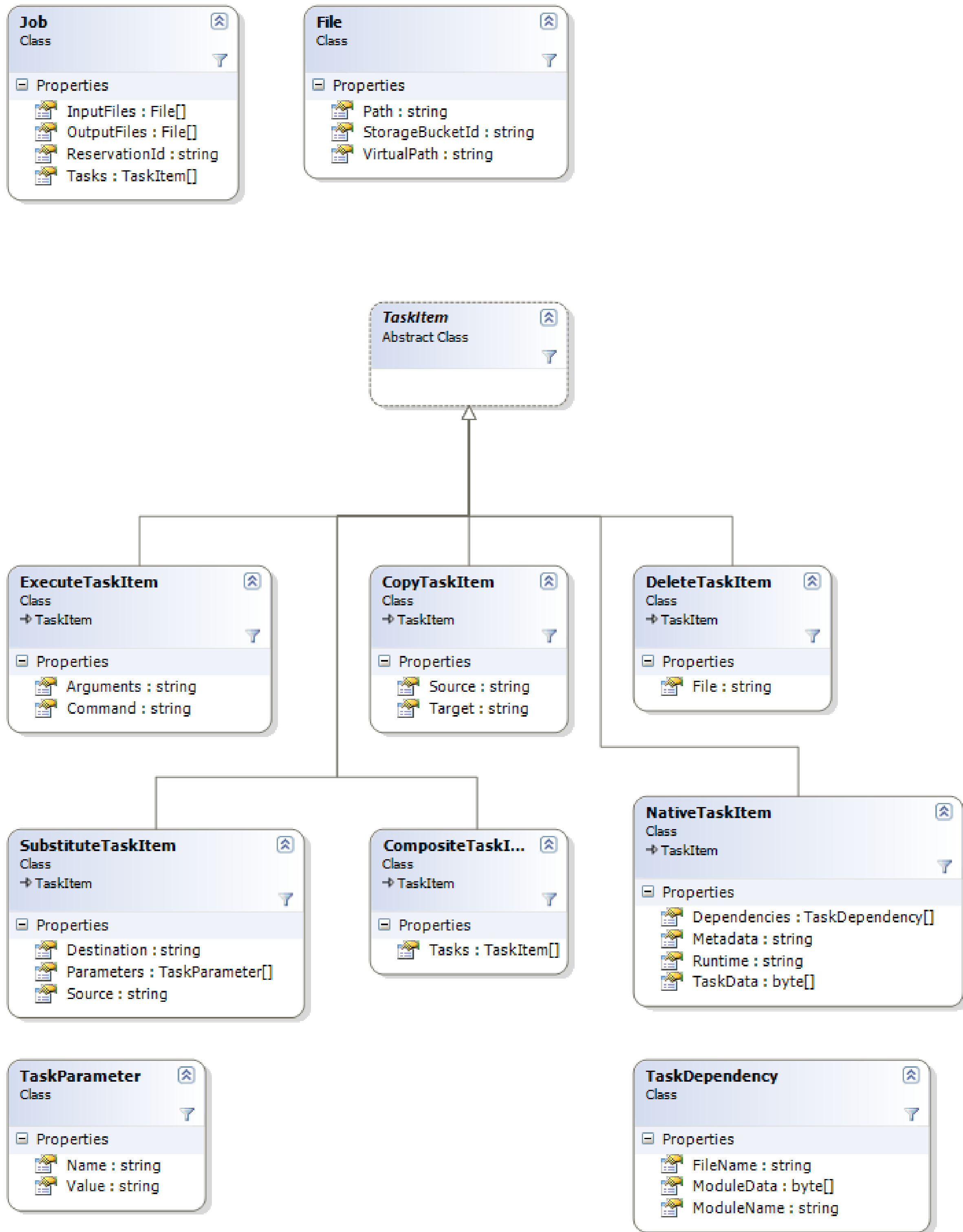

Figure 4 - Web Service Job Object Model

Copyright © 2011Manjrasoft Pty Ltd.

- **DeleteTaskItem**: this object is mapped to a *DeleteTask* instance. The *DeleteTask* class allows users to delete a single file without the use of wildcards. This restriction is limiting but we are working on providing a safe way to include wildcards and extend the functionality provided by this command.
- **ExecuteTaskItem**: this object is mapped to an *ExecuteTask* instance. The *ExecuteTask* class allows users to execute any command that can be issued by the shell. Since it is not possible to upload files through the web service interface the effectiveness of this task is quite limited and is restricted to executing pre-existing command or applications installed on the remote node.
- **SubstituteTaskItem**: this object is mapped to a *SubstiuteTask* instance. The *SubstituteTask* class allows users to substitute to create a file from a template file with the assignment defined in the collection of parameters passed together with the task.
- **CompositeTaskItem**: this object is mapped to a *CompositeTask* instance. The *CompositeTask* class defines a task that is made out of other tasks, and constitutes a flexible way to execute and reuse complex tasks. The *CompositeTask* type exposes an list of tasks. This list can contain any task type implementing the *ITask* interface. When used through the web service interface it is only possible to create compose tasks that are made out of the tasks accessible via the web service interface (this means: *ExecuteTask*, *CopyTask*, *DeleteTask*, *SubstituteTask*, *CompositeTask*).

Moreover, besides these common tasks the **NativeTaskItem** class allows developers to supply through the web service a task implementation for a specific runtime. This capability gives a high degree of freedom in terms of operational logic that can be expressed for the execution of the task. By using this task, virtually all the *Aneka.Tasks.ITask* instances that can be submitted through the client in-process APIs can be submitted through the web service. The design goal of the *NativeTaskItem* class is to provide a way to execute object code that requires a specific runtime (ad es: Java) and in order to do so exposes the following properties:

- **Dependencies**: an array of *TaskDependency* instances that are used to submit all the library dependencies that are required for the execution of the task (ad ex: .class files in the case of Java). A *TaskDependency* instance allows wrapping a binary array storing the file content, a string representing the name under which the file will be saved, and another string containing the name of the module. In the case of the .NET runtime these properties are set as follows:
    - *FileName*: name of the dll/exe file that represent the assembly.
    - *ModuleData*: content of the assembly in bytes.
    - *ModuleName*: assembly name.
- **Runtime**: a string that identifies the specific runtime environment required by the task at the moment the only value supported is CLI#2.0, which identifies .NET managed code.
- **Metadata**: a string representing the command arguments that are passed to the interpreter or runtime environment and that specify how to execute the native

Copyright © 2011Manjrasoft Pty Ltd.

task. At the moment, since the only runtime supported is .NET, this value should contain the assembly qualified type name of the class implementing *Aneka.Tasks.ITask*.

- **TaskData**: this is an array of bytes representing the serialized instance of the task to execute that will be passed to the runtime environment or interpreter. In the case of .NET is the binary serialized form of a managed type.

This class is mapped to the *Aneka.Tasks.BaseTasks.RuntimeNativeTask* class, which unwraps and copies the dependencies into the local execution directory, deserializes the binary instance, and executes it. Basically this class does what already the task execution service does, so why it has been designed? As already said, the *NativeTaskItem* class allows specifying native tasks which could have been directly turned into *ITask* instances and wrapped around *AnekaTask* objects without the use of the *RuntimeNativeTask* class. This implementation has a major drawback, which is the fact that the libraries defining the *ITask* submitted by the user need to be loaded into the process of the web service. This is not only unnecessary but also dangerous since the user could define *ITask* instances that execute code in the constructor. The execution of this code occurs outside a proper sandbox and might endanger the web service itself. By using the *RuntimeNativeTask* the task instance is only unwrapped in the remote execution sandbox.

Even though this is a primary requirement, the reason of using *RuntimeNativeTask* is to support different runtime environment in the future, thus allowing applications implemented in other programming languages and environments (i.e. Java) to directly serialize object instances in process and submit them for execution through the language agnostic web service. At the moment there is no other runtime supported than .NET/Mono 2.0, therefore this capability is only potential.

5.2 File Management

Task item operate on files either they are shared, input, or output files. Except for the case of the *NativeTaskItem* all the pre-built task items operate on files. The web service object model uses the File class to wrap the required information to locate and utilize a file within Aneka. This class is mapped by the Web Service to an instance of the *Aneka.Data.FileData* class. The *File* class exposes the following properties:

- **Path**: this is a string that is used to represent the path to the file. For input and shared files this value is the location of origin of the file with respect to the storage repository where it is stored. For example in the case of a remote FTP, which is the most common scenario, this value represents the relative path to the user root in the FTP server where the file is stored. For output files this value represents the final location where the file will be stored with respect to the target storage repository. Again, in case of an FTP storage server this value will be the location where the file will be uploaded in the FTP with respect to the user root.
- **VirtualPath**: this is a string that represents the path to the file on the remote execution node. In case of input and shared files this will be the location where these files are saved (in general this property only stores the name of the file). In

Copyright © 2011Manjrasoft Pty Ltd.

case of output files this value represents the location where the execution service is expected to find these files.

- **StorageBucketId**: this is a string that represents the key mapping the configuration data of the corresponding storage server that relates to the file (origin in case of shared and input files, destination in case of output files).

The interface of the Web Service does not allow developer to upload files through the web service as they would normally do with the client APIs of the *Task Programming Model*. Instead, input files and shared files are expected to be located in (and output files are expected to be moved to) remote storage servers. In this way, it is responsibility of the *Storage Service* installed in Aneka to pull in and pull out files and not a task performed by the web service. This allows a light implementation of the web service, which can use a simplified version of the client APIs, without active components, and does not need special storage for temporarily saving input and output files.

Given this architecture developers have to specify and inform Aneka about how to collect and store files. This is done by providing the proper information in the *ApplicationSubmissionRequest.Metadata* property group. Developers have to create a group of properties called *StorageBuckets* and define a group (together with the required properties) for each storage server that is used by the application. The following code shows how to configure an application submission request with two different storage servers:

```
ApplicationSubmissionRequest request = new ApplicationSubmissionRequest();
// initialize the main group of properties.
request.Metadata = new PropertyGroup();

// create the "StorageBuckets" group and add it to the metadata.
PropertyGroup buckets = new PropertyGroup("StorageBuckets");
buckets.Groups = new PropertyGroup[2];
request.Metadata.Groups = new PropertyGroup[1];
request.Metadata.Groups[0] = buckets;

// create the "InputFTP" group wrapping the properties required
// to access a remote ftp server.
PropertyGroup inputFtp = new PropertyGroup("InputFTP");
inputFtp.Properties = new Property[5];

Property ftp = new Property();
ftp.Name = "StorageType";
ftp.Value = "FTP";
inputFtp.Properties[0] = ftp;

Property user = new Property();
user.Name = "User";
user.Value = "anonymous";
inputFtp.Properties[1] = user;

Property password = new Property();
password.Name = "Password";
password.Value = "anonymous";
inputFtp.Properties[2] = password;

Property host = new Property();
```

Copyright © 2011Manjrasoft Pty Ltd.

```
host.Name = "Host";
host.Value = "124.32.33.35"; // DNS name or IP
inputFtp.Properties[3] = host;

Property port = new Property();
port.Name = "Port";
port.Value = "21";
inputFtp.Properties[4] = port;

buckets.Groups[0] = inputFtp;

// create an S3 storage bucket for the output files
PropertyGroup outputS3= new PropertyGroup("OutputS3");
outputS3.Properties = new Property[3];

Property s3 = new Property();
s3.Name = "StorageType";
s3.Value = "S3";
outputS3.Properties[0] = s3;

Property accessKey = new Property();
accessKey.Name = "AWSAccessKeyId";
accessKey.Value = <your access key id:string>;
outputS3.Properties[1] = user;

Property bucket= new Property();
bucket.Name = "Bucket";
bucket.Value = <your bucket name:string>;
outputS3.Properties[2] = bucket;

buckets.Groups[1] = outputS3;
```

Once this information is entered in order to reference a storage bucket in a File instance we can simply set the *File.StorageBucketId* property to the name of the *PropertyGroup* mapping the configuration of the bucket to use. Here follows an example on how to set these values for input and output files:

```
File shared1 = new File();
shared1.Path = "/shared/model.dat";
shared1.VirtualPath = "model.dat";
shared1.StorageBucketId = "InputFtp";

File input1 = new File();
input1.Path = "/input-samples/sample.1.dat";
input1.VirtualPath = "sample.1.dat";
input1.StorageBucketId = "InputFtp";

File output1 = new File();
output1.Path = "result.1.dat";
output1.VirtualPath = "result.1.dat";
output1.StorageBucketId = "OutputS3";
```

With this information the storage server will automatically pull in the input files from remote FTP server and push the output file to the remote S3 storage bucket.

Copyright © 2011Manjrasoft Pty Ltd.

6 Storage Service Configuration

In order to operate seamlessly with the web service the Aneka Cloud needs to be able to autonomously pull in and out files from the internal Storage Service. By default this capability is not configured and needs to be activated by modifying the Spring configuration file of the container hosting the Storage Service.

The default implementation of the *Storage Service* (Aneka.Storage.dll) is based on the concept of factories for handling the file transfer between nodes. A file transfer factory provides facilities for creating server and client components that are used in for transferring files. In this way, it decouples the management and the interaction with the Aneka runtime from the details of the specific file transfer channel. By default the *Storage Service*, uses the *FTPFileTransferFactory* (Aneka.Data.Ftp.dll), which allows the creation of client and server components exchanging files through the FTP protocol. When the service starts it creates a server component to manage the storage made available through the server, and dynamically provides libraries to each node acting as a client for the storage (execution nodes and client application node).

When it comes to autonomously importing and exporting files from and to a remote storage server, the Storage Service itself becomes a client and therefore it needs to be configured to properly access the remote server. Since the entire design of the Storage Service is unaware of the specific protocol used to move files, this property has also been maintained in the case of communication with external repositories. Therefore, it needs to dynamically select the appropriate client module according to the type remote storage. Again, this capability has been implemented by leveraging file transfer factories. The *Storage Service* maintains a dictionary that maps the name of the *StorageType* property in the storage bucket to the corresponding implementation of file transfer factory implementation that can be used to obtain a client component for the given storage type. When there is a file to import from a remote server (or to export to a remote server) the Storage Services looks up the value of the storage type of the corresponding storage bucket and uses it to retrieve the file transfer factory implementation that is used to create a client component configured to access the remote server where the file is located (or where it needs to be exported). This operation is possible if and only if the dictionary actually contains the metadata of the factory for a given storage type. This is the kind of information that needs to be configured in order to enable the Storage Service to autonomously import and export files.

In order to configure such behaviour it is necessary to set the *FileTransferFactoryMap* property of the *Storage Service*. Currently there is no GUI support for such feature and the configuration needs to be done directly by modifying the spring file. This is a dictionary that maps a string to instance of type *Aneka.Data.IFileTransferFactory* (Aneka.Data.dll). In order to configure the Storage Service for interacting with remote FTP servers we need to add an entry that maps the FTP storage type to the already existing *FtpFileTransferFactory* object used to creating components for the internal storage. The following XML block shows how to configure such feature into the <object /> node that relates to the Storage Service.

```
<property name="FileTransferFactoryMap">
  <dictionary key-type="string"
              value-type="Aneka.Data.IFileTransferFactory,Aneka.Data">
```

```
    <entry key="FTP" value-ref="FtpFileTransferFactory" />
  </dictionary>
</property>
```

The current implementation of Aneka features only the implementation of the file transfer factory for the FTP protocol and for S3 (only the client module). Should developers provide their own implementation of the factory; this can be added into this map and used seamlessly with the existing infrastructure. Possible alternative to these implementations might be:

- *SSH / SCP*
- *SFTP*
- *Shared file system support*

The interfaces that define the object model of file transfer factories are declared into the *Aneka.Data* namespace (Aneka.Data.dll) and are: *IFileTransferFactory* (factory component), *IFileChannelController* (server component), *IFileHandler* (client component).

7 Advanced Topics and Technical Details

This section addresses some advanced aspects of the use of the *Task Submission Web Service* for submitting and monitoring applications on Aneka. Web services are use the HTTP protocol as main transport layer and therefore are generally designed as stateless services. This property makes the management of Aneka applications and more precisely the monitoring of their execution more complex. In this section we will address the most important aspects that developers need to take into account while operating through the web service.

7.1 Disconnected Mode vs. Connected Mode

Web services are stateless entities (or better, they should be designed like that =)). This means that they do not have active components (i.e. asynchronous threads) that operate in background without the user intervention. On the contrary, every operation performed by a web service is triggered by a user request and should be completed within the context of the web service method call, so that the outcome of the call can be returned to the client application. This is also the way in which the Task Web Service works because it is designed to serve a multitude of clients and its execution needs to be quick and light.

The Aneka client libraries for the in-process management of distributed applications (i.e. *AnekaApplication<AnekaTask,TaskManager>*) rely on active components that constantly interact with the Aneka Cloud through a persistent connection. This approach makes possible the use of events to signal the client process when some conditions on the server side are verified. Events are used to signal the client process the following:

- Task state transitions.
- Task failure and completion.
- Application failure and completion.

These events are not available through the Task Web Service and web service clients discover get to know these server side events when they actively make a call to a web service method. Therefore, i might happen one of the following:

- Clients try to submit/query jobs for an application that does not exist anymore because it is failed or completed.
- Clients try to obtain the full information of a job that has completed or that is failed.

The first case might still happen in the connected scenario but events on the *AnekaApplication* help developers to react in time. The second case, instead, is more peculiar and becomes evident only in the case of the Web Service scenario.

This is due to the particular way in which the Scheduling Service manages the completed, failed, and aborted WorkUnit instances (of almost any model, *Task Programming Model* and *Thread Programming Model* included). As soon as one *WorkUnit* transits into one of the following states:

- *Failed*
- *Aborted*
- *Completed*

After the first query of its status the information about its binary instance is removed from the application store and its metadata is persisted into the accounting store if there is one. Now, the question is:

Who does make the first query?

In the case of the in-process client libraries, the *TaskManager* has a query thread that constantly polls the *Scheduling Service* and asks for the state of the pending tasks. Once a tasks transits into one of these three states, the instance data of the task is collected by the query thread and the appropriate event on the client side is fired (*WorkUnitFailed* in case of *Failed* or *Aborted*, *WorkUnitFinished* in case of *Done/Completed*). In this way developers can capture the instance of the task and collect the desired information from the deserialized object. After such operation, the information on the instance of the task is deleted by the scheduler because there is no need to have such data anymore.

This behaviour might be acceptable even in the case of the web service. In reality, being the web service a disconnected client (with no events) it might be possible that the information about a task completed or failed gets lost unless a proper management is done in the web service client code. In practice, this means developing another *TaskManager* that uses the web service as active connection to Aneka. This is obviously something to avoid because it does not make a proper use of the web service.

In order to avoid this situation the *Scheduling Service* and the *Execution Service* have been modified to support the capability of:

- Keeping track of the binary instance of completed tasks.
- Storing the binary instance on a persistent storage provided by the user.

These are two solutions that provide a different workaround to the problem. The first one is implemented in the scheduler and it allows triggering this behavior per application. The second one is implemented in the executor nodes and again can be activated for a specific

application. While the first option requires configuration changes on both the Scheduling Service and the client application, the second one is simply triggered by the configuration of the application.

7.1.1 Task Persistence in the Scheduling Service

This feature controls the capability of the scheduler of preserving the binary instance of a terminated task until the application is not completed. This feature is activated by setting the *KeepAliveMode* property of the *Metadata* property of the service to true. If this flag is set, the scheduler will keep the serialized instances of failed/aborted/completed tasks for those applications that require it. Therefore, in order to obtain the instance of a task through the web service it is necessary to perform the following steps:

- Configure the *Task Scheduling Service* with the *KeepAliveMode* set to true.
- Configure the application submission in order to require the persistence of work unit.

The first step is done by entering a *Property(Name="KeepAlive", Value="true")* into the *Metadata* property group exposed by the service. This feature is not supported through the *Management Studio* and needs to be configured **by directly changing the Spring.config.xml file** of the container instance hosting the *Task Scheduling Service*. The following excerpt of the *Spring.config.xml* file shows how to configure such a property:

```
<object name="KeepAliveMode" type="Aneka.Property, Aneka.Util">
   <property name="Name" value="KeepAliveMode"/>
   <property name="Value" value="True"/>
</object>
<object name="Metadata" type="Aneka.PropertyGroup, Aneka.Util">
  <property name="Name" value="Metadata"/>
  <property name="Properties">
     <list>
        <ref object="KeepAliveMode"/>
     </list>
  </property>
</object>
<object name="TaskSchedulingHandler"
       type="Aneka.Tasks.Scheduler.TaskSchedulingHandler,
            Aneka.Tasks.Scheduler" />
<object name="TaskSchedulingAlgorithm"
      type="Aneka.Scheduling.Algorithms.Independent.FIFOSchedulingAlgorithm,
           Aneka.Scheduling" />
<object name="TaskScheduler"
       type="Aneka.Scheduling.Service.IndependentSchedulingService,
            Aneka.Scheduling">
  <property name="ApplicationStore" ref="ApplicationStore" />
  <property name="SchedulingHandler" ref="TaskSchedulingHandler" />
  <property name="SchedulerAlgorithm" ref="TaskSchedulingAlgorithm" />
  <property name="Metadata" ref="Metadata" />
</object>
```

In the XML excerpt we configure the *Task Scheduling Service* with an object named *TaskScheduler* and we set the *Metadata* property group previously configured with *KeepAliveMode* property.

Copyright © 2011Manjrasoft Pty Ltd.

The second step is performed during application submission and involves setting a property into the Metadata property group of the *ApplicationSubmissionRequest* instance. As described in the following code sample:

```
// Create an application submission request instance and the metadata property
group.
ApplicationSubmissionRequest request = new ApplicationSubmissionRequest();
request.Metadata = new PropertyGroup();
request.Metadata.Name = "Metadata";

// Create the property group for the general properties...
request.Metadata.Groups = new PropertyGroup[1]; // put 2 if you need to add
storage buckets.
PropertyGroup general = new PropertyGroup();
general.Name = "General";
general.Properties = new Property[1];
request.Groups[0] = general;

// Create the keep alive property and add it to the general group.
Property keepAlive = new Property();
keepAlive.Name = "KeepAlive";
keepAlive.Value = Bool.TrueString; // "True"
general.Properties[0] = keepAlive;
```

In this way we have configured entered the following setting: *General.KeepAlive* = *true*, which together with the configuration made for the scheduler allows to keep track of the persisted task instances until the application completes.

7.1.2 Task Persistence in the Execution Service

This feature provides an alternative solution to the problem of the lost units and does not involve any change to the configuration of the *Scheduling Service* neither a slowdown of its performance. The disadvantage is that it provides a method to retrieve the information we want but not as seamlessly as we can do by using the previous approach. This solution is completely controlled by the client configuration and involves the use of the *General.InstanceBucketId* property. This is the name of the storage bucket to use for saving the persisted instances (if possible) of the terminate units on storage server provided by the user.

The following excerpt of code shows how to configure the application submission request in order to save the binary instance of the completed tasks into the previously configured S3 storage bucket named *"OutputS3"*.

```
// Create an application submission request instance and the metadata property
group.
ApplicationSubmissionRequest request = new ApplicationSubmissionRequest();
request.Metadata = new PropertyGroup();
request.Metadata.Name = "Metadata";

// Create the property group for the general properties...
request.Metadata.Groups = new PropertyGroup[2]; // 2 one for "General" and
another for "StorageBuckets".

// here the code of the configuration of the storage bucket follows...
```

Copyright © 2011Manjrasoft Pty Ltd.

```
PropertyGroup general = new PropertyGroup();
general.Name = "General";
general.Properties = new Property[1];
request.Groups[1] = general;

// Create the bucket id property and add it to the general group.
Property bucketId = new Property();
bucketId.Name = "InstanceBucketId";
bucketId.Value = "OutputS3";
general.Properties[0] = bucketId;
```

By using this approach we can allow clients to later retrieve these instances at their own convenience. By querying the web service they will only know that the task has been terminated.

7.1.3 Observations

Keeping track of the serialized instance of completed tasks is important if and only if the information in has a meaningful value. This is in general the case of native tasks that store the outcome of their execution into instance properties. Therefore, only in this specific case (which generally implies a connected mode) it is important to activate one of these two features. As we have already pointed out, the first one allows for a completely transparent management of task instance through the web service but put some burden in terms of local storage on the scheduler, while the second one is more efficient but requires proper programming.

It is worth to say that in a web service scenario it is a good practice to store all the valuable information coming out from the execution of a task into output files. This is because the web service is designed to work seamlessly across different platforms and technologies while the binary serialization is limited to the .NET world.

8 Web Service Configuration and Deployment

The web service does not feature a complete installation of Aneka but it only provides a web interface to the *Task Programming Model* and provides facilities for submitting tasks, monitor their execution, and collect their result in terms of instances or output files. Therefore, in order to properly operate the web service needs to be configured to point to a specific installation of Aneka and this is information is provided by specifying the URI of the index node of the Cloud in the *<appSettings />* section of the *Web.config* file as shown below:

```
<appSettings>
   <add key="IndexNodeUri" value="IndexNodeUri" />
</appSettings>
```

By default the service is configured in order to point to a local installation on the default port, but any valid Aneka uri can be provided. The web service will the act towards the index node on behalf of the user that makes the web request and by using the credentials provided together with it.

Copyright © 2011Manjrasoft Pty Ltd.

8.1 Installation on Windows Operating Systems

The current release of Aneka (version: 2.0.2.0) provides a Windows Installer Package that can be used to deploy the Task Submission Web Service on any computer that has installed "Internet Information Services (IIS)". The package has been compiled with Visual Studio 2005 therefore in case of Windows Vista/ Windows 7 it is necessary to activate the retro compatibility feature by doing the following:

- Open the **Control Panel → Programs** options tab.
- Select **Turn Windows features on or off**.
- Expand the **Internet Information Services** options group.
- **Check all the options**.

Obviously, you need to have IIS installed on the target machine. This component can be installed by looking in the Windows components.

8.2 Installation on Unix-based Systems with Mono

At the moment there is no default installation package for *Mono*. The simplest thing to do is to manually configure the web service on *XSP* or *Apache/mod_mono*. In order to do that it is necessary to copy the content of the IIS virtual directory created for the web service into a directory served by the web server and properly configure the directory so that the web service will be processed as active content.

The ASP.NET Mono Page can be taken as a starting pointer on how to deploy ASP.NET web applications on Mono.

9 Conclusions

In this document we have briefly addresses how to use the Task Submission Web Service for the submission, management, and monitoring of applications based on the Task Programming Model. The web service allows performing the following operations:

1. Authentication on behalf of the client to Aneka.
2. Creation of an application based on the *Task Programming Model*.
3. Submission of jobs (i.e. *AnekaTask* instances) either by providing native tasks or using predefined template tasks.
4. Monitoring the status of jobs, retrieving their results, and aborting them.
5. Monitoring the status of an application and aborting its execution.

It is also possible to bind an existing application and perform the operation 2,3,4, and 5 in the previous list.

Another aspect that has been addressed in this article is the management of files. The task web service does not allow the direct submission of shared and input files for jobs neither allows the download of output files. Instead, it requires client applications to store and retrieve all these files by using remote storage servers (at the moment FTP and S3 storage types are supported). This allows the web service implementation to be simple light and then fast.

Copyright © 2011Manjrasoft Pty Ltd.

We have also discussed the implications of the disconnected behaviour of the Web Services with reference to the collection of the outcome of task execution in the form of the serialized task instance. Finally, we have given guidance on how to deploy the web service on different web hosting environments such *as IIS* / *Mono XPS* / *Apache mod_mono*.

Copyright © 2011Manjrasoft Pty Ltd.