Chapter 7 — High-Throughput Computing: Task Programming

Task computing is a wide area of distributed system programming encompassing several different models of architecting distributed applications, which, eventually, are based on the same fundamental abstraction: the task. A task generally represents a program, which might require input files and produce output files as a result of its execution. Applications are then constituted by a collection of tasks. These are submitted for execution and their output data is collected at the end of their execution. The way in which tasks are generated, the order in which they are executed, or whether they need to exchange data, differentiate the application models falling under the umbrella of task programming.

This chapter characterizes the abstraction of a task and provides a brief overview of the distributed application models that are based on the task abstraction. The Aneka Task Programming Model is taken as a reference implementation to illustrate the execution of Bag of Tasks (BoT) applications on a distributed infrastructure.

7.1 Task Computing

Organizing an application in terms of tasks is the most intuitive and common practice for developing parallel and distributed computing applications. A task identifies one or more operations that produce a distinct output and that can be isolated as a single logical unit. In practice, a task is represented as a distinct unit of code, or a program, that can be separated and executed in a remote runtime environment. Programs are the most common option for representing tasks, especially in the field of scientific computing, which has leveraged distributed computing for its computational needs.
Multi-threaded programming is mainly concerned with providing a support for parallelism within a single machine. Task computing provides distribution by harnessing the compute power of several computing nodes. Hence, the presence of a distributed infrastructure is explicit in this model. Historically, the infrastructures that have been leveraged to execute tasks are clusters, supercomputers, and computing Grids. Now Clouds have emerged as an attractive solution to obtain a huge computing power on-demand for the execution of distributed applications. In order to achieve it, a suitable middleware is needed. A reference scenario for task computing is depicted in Figure 7.1.

![Figure 7.1. Task Computing Scenario.](image)

The middleware is a software layer that enables the coordinated use of multiple resources, which are drawn from a Data Center or geographically distributed networked computers. A user submits the collection of tasks to the access point(s) of the middleware, which will take care of scheduling and monitoring the execution of tasks. Each computing resource provides an appropriate runtime environment, which may vary from implementation to implementation (a simple shell, a sandboxed environment, or a virtual machine). Task submission is normally done by using the APIs provided by the middleware, whether they are a web or a programming language interface. Appropriate APIs are also provided to monitor task status and collect their results upon the completion.

As task abstraction is general, there exist different models of distributed applications falling under the umbrella of task computing. Despite this, it is possible to identify a set of common operations that the middleware needs to support the creation and execution of task based applications. They are:

- Coordination and scheduling of tasks for execution on a set of remote nodes.
- Movement of programs to remote nodes and management of their dependencies.
- Creation of an environment for execution of tasks on the remote nodes.
• Monitoring the task’s execution and inform the user about its status.
• Access to the output produced by the task.

Models for task computing may differ in the way tasks are scheduled which in turn depends on whether tasks are interrelated or they need to communicate amongst themselves.

7.1.1 Characterizing a Task

A task is a general abstraction identifying a program or a combination of programs that constitute a computing unit of a distributed application with a tangible output. It represents a component of an application that can be logically isolated and executed separately. Distributed applications are composed of tasks, whose collective execution and interrelations define the nature of the applications. A task can be represented by different elements:

• A shell script composing together the execution of several applications.
• A single program.
• A unit of code (a Java/C++/.NET class) that executes within the context of a specific runtime environment.

A task is generally characterized by input files, executable code (programs, shell scripts, etc.), and output files. In many cases the common runtime environment in which tasks executes, is represented by the operating system or an equivalent sandboxed environment. A task may also need specific software appliances on the remote execution nodes in addition to the library dependencies that can be transferred to the node.

Some distributed applications may have additional constraints. For example, distributed computing frameworks that present the abstraction of task at programming level, by means of a class to inherit or an interface to implement, might require additional constraints (i.e. compliance to the inheritance rules) but also a richer set of features that can be exploited by developers. According to the specific model of application, tasks might have dependencies.

7.1.2 Computing Categories

According to the specific nature of the problem, different categories for task computing have been proposed over time. These categories do not enforce any specific application model but provide an overall view of the characteristics of the problems. They implicitly impose requirements on the infrastructure and the middleware. Applications falling in this category are: High-Performance Computing (HPC), High-Throughput Computing (HTC), and Many Tasks Computing (MTC).

High Performance Computing (HPC)

High Performance Computing is the use of distributed computing facilities for solving problems that need large computing power. Historically, supercomputers and clusters are specifically designed to support HPC applications that are developed to solve grand
challenging problems in science and engineering. The general profile of HPC applications is constituted by a large collection of compute intensive tasks which needs to be processed in a short period of time. It is common to have parallel and tightly coupled tasks, which require low latency interconnection network to minimize the data exchange time. The metrics to evaluate HPC systems are FLOPS (now Tera-FLOPS or even Peta-FLOPS), which identify the number of floating point operations per second that a computing system can perform.

**High Throughput Computing (HTC)**

High Throughput Computing is the use of distributed computing facilities for applications requiring large computing power over a long period of time. HTC systems need to be robust and reliably operate over a long time scale. Traditionally, computing Grids composed of heterogeneous resources (clusters, workstations, and volunteer desktop machines), have been used to support HTC. The general profile of HTC applications is that are made up of a large number of tasks, whose execution can last for a considerable amount of time (i.e. weeks or months). Classical examples of such applications are scientific simulations or statistical analyses. It is quite common to have independent tasks that can be scheduled in distributed resources as they do not need to communicate. HTC systems measure their performance in terms of jobs completed per month.

**Many Tasks Computing (MTC)**

Many Tasks Computing [61] model started receiving attention recently and it covers a wide variety of applications. It aims to bridge the gap between HPC and HTC. MTC is similar to High Throughput Computing, but it concentrates on the use of many computing resources over a short period of time to accomplish many computational tasks. In brief, MTC denotes high-performance computations comprising multiple distinct activities, coupled via file system operations. What characterizes MTC is the heterogeneity of tasks that might be of considerably different nature: tasks may be small or large, single processor or multi-processor, compute-intensive or data-intensive, static or dynamic, homogeneous or heterogeneous. The general profile of MTC applications include loosely coupled applications that are generally communication-intensive but not naturally expressed using message passing interface commonly found in HPC, drawing attention to the many computations that are heterogeneous but not embarrassingly parallel. Given the large amount of tasks commonly composing MTC applications, any distributed facility with a large availability of computing elements is able to support MTC. Such facilities include: supercomputers, large clusters, and emerging Cloud infrastructures.

**7.1.3 Frameworks for Task Computing**

There are several frameworks that can be used to support the execution of task-based applications on distributed computing resources including Clouds. Some popular software
systems supporting task computing framework are: Condor [5], Globus Toolkit [12], Sun Grid Engine (SGE) [13], BOINC [14], Nimrod/G [164], and Aneka.

Architecture of all these systems is similar to the general reference architecture depicted in Figure 7.1. They consist of two main components: a scheduling node (one or more) and worker nodes. The organization of the components of the system may vary. For example, multiple scheduling nodes can be organized in hierarchical structures. This configuration is quite common in case of the middleware for computing Grids, which harness a variety of distributed resources from one or more organizations or sites. Each of these sites may have their own scheduling engine, especially if the system contributes to the Grid but also serves local users.

A classic example is the cluster setup where the system might feature an installation of Condor or SGE for batch job submission; these services are generally used locally to the site, but the cluster can be integrated into a larger Grid where meta-schedulers such as GRAM (Globus Resource Allocation Manager)\(^\text{38}\) can dispatch a collection of jobs to the cluster. Other options include the presence of gateway nodes that do not have any scheduling capabilities and simply constitute the access point to the system. These nodes have indexing services that allow users to identify the available resources in the system, its current status, and the available schedulers. For worker nodes, they generally provide a sandboxed environment where tasks are executed on behalf of a specific user or within a given security context limiting the operations that can be performed by programs such as file system access. File staging is also a fundamental feature supported by these systems. Clusters are normally equipped with shared file systems and parallel I/O facilities. Grids provide users with different staging facilities such as credential access to remote worker nodes or automated staging services that transparently move files from user local machine to remote nodes.

Condor is probably the most widely used and long-lived middleware for managing clusters, idle workstations, and a collection of clusters. Condor-G is a version of Condor supporting the integration with Grid computing resources, such as those managed by Globus. Condor supports common features of batch queuing systems along with the capability of checkpointing of jobs and management of overload nodes. It provides a powerful job-resource matching mechanisms, which schedule jobs only on resources having the appropriate runtime environment. It can handle both serial and parallel jobs on a wide variety of resources. It is used by hundreds of organizations in industry, government, and academia to manage infrastructures ranging from a handful to a well over thousands of workstations.

Sun Grid Engine (SGE), now Oracle Grid Engine, is a middleware for workload and distributed resource management. Initially developed to support the execution of jobs on clusters, Sun Grid Engine integrated additional capabilities and now is able to manage heterogeneous

\(^{38}\) Globus Resource Allocation Manager is a software component of the Globus Toolkit that is in charge of locating, submitting, monitoring, and cancelling jobs in Grid computing systems.
resources and constitutes a middleware for Grid computing. It supports the execution of parallel, serial, interactive, and parametric jobs and features advanced scheduling capabilities such as budget-based and group-based scheduling, scheduling applications that have deadlines, custom policies, and advance reservation.

The Globus Toolkit is a collection of technologies enabling Grid Computing. It provides a comprehensive set of tools for sharing computing powers, databases, and other services across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit features software services, libraries, and tools for resource monitoring, discovery, and management and also security and file management. The Globus Toolkit addresses core issues of Grid computing: the management of a distributed environment composed by heterogeneous resources spanning across different organizations with all this condition implies in terms of security and interoperation. In order to provide a valid support for Grid computing in such scenario the toolkit defines a collection of interfaces and protocol for interoperation that enable different system to integrate with each other and expose resources outside their boundaries.

Nimrod/G [164] is a tool for automated modeling and execution of parameter sweep applications (parameter studies) over global computational Grids. It provides a simple declarative parametric modeling language for expressing parametric experiments. A domain expert can easily create a plan for a parametric experiment and use the Nimrod/G system to deploy jobs on distributed resources for execution. It has been used for a very wide range of applications over the years, ranging from quantum chemistry to policy and environmental impact. Moreover, it uses novel resource management and scheduling algorithms based on economic principles. Specifically, it supports deadline and budget constrained scheduling of applications on distributed Grid resources to minimize the execution cost and at the same deliver results in timely manner.

BOINC (Berkley Open Infrastructure for Network Computing) is framework for volunteer and Grid computing. It allows turning desktop machines into volunteer computing nodes that are leveraged to run jobs when such machines become inactive. BOINC is composed by two main components: the BOINC server and the BOINC client. The former is the central node keeping track of all the available resources and scheduling jobs, while the latter is the software component that is deployed on desktop machines and that creates the BOINC execution environment for job submission. Given the volatility of BOINC clients, BOINC supports job check-pointing and duplication. Even if mostly focused on volunteer computing, BOINC systems can be easily setup to provide a more stable support for job execution by creating computing Grids with dedicated machines. In order to leverage BOINC it is necessary to create an application project. When installing BOINC clients, users can decide the application project to which they want to donate the CPU cycles of their computer. Currently, there are several projects, ranging from medicine to astronomy and cryptography, are running on the BOINC infrastructure.
7.2 Task-based Application Models

There exist several models that are based on the concept of task as the fundamental unit for composing distributed applications. What makes them different is the way in which tasks are generated, the relations they have with each other, the presence of dependencies, or other conditions, for example a specific set of services in the runtime environment, that have to be met. In this section, we quickly review the most common and popular models based on the concept of task.

7.2.1 Embarrassingly Parallel Applications

Embarrassingly parallel applications constitute the most simple and intuitive category of distributed applications. As already discussed in the previous chapter, embarrassingly parallel applications are constituted by a collection of tasks that are independent from each other and that can be executed in any order. The tasks might be of the same type or of different nature and they do not need to communicate among themselves.

This category of applications is supported by the majority of the frameworks for distributed computing. Since tasks do not need to communicate, there is a lot of freedom for the way in which they are scheduled. Tasks can be executed in any order and there is no specific requirement for tasks to be executed at the same time. Therefore, scheduling of these applications is simplified and mostly concerned with the optimal mapping of tasks to available resources. Frameworks and tools supporting embarrassingly parallel applications are the Globus Toolkit, BOINC, and Aneka.

There are several problems that can be modeled as embarrassingly parallel. These include image and video rendering, evolutionary optimization, model forecasting, etc. In case of image and video rendering the task is represented by the rendering of a pixel (more likely a portion of the image) or a frame respectively. For evolutionary optimization metaheuristics, a task is identified by a single run of the algorithm with a given parameter set. The same applies to model forecasting applications. In general, scientific applications constitute a considerable source of embarrassingly parallel applications, even though they mostly fall into the more specific category of parameter sweep applications.

7.2.2 Parameter Sweep Applications

Parameter sweep applications are a specific class of embarrassingly parallel applications whose tasks are identical in their nature and differ only by the specific parameters used to execute them. Parameter sweep applications are identified by a template task and a set of parameters. The template task defines the operations that will be performed on the remote node for the execution of tasks. The template task is parametric and the parameter set identifies the combination of variables whose assignments specialize the template task into a specific instance. The combination of parameters together with their range of admissible
values identifies the multi-dimensional domain of the application and each point in this
domain identifies a task instance.

Any distributed computing framework providing support for embarrassingly parallel
applications can also support the execution of parameter sweep applications, since the tasks
composing the application can be executed independently from each other. The only
difference is that the tasks that will be executed are generated by iterating over all the
possible and admissible combinations of parameters. This operation can be performed
frameworks natively or tools that are part of the distributed computing middleware. For
example, Nimrod/G is natively designed to support the execution of parameter sweep
applications and Aneka provides client-based tools for visually composing a template task,
define parameters, and iterate over all the possible combinations of such parameters.

There is a plethora of applications that fall into this category. Mostly, they come from the
scientific computing domain: evolutionary optimization algorithms, weather forecasting
models, computational fluid dynamics applications, Monte Carlo methods, and many others.
For example, in the case of evolutionary algorithms it is possible to identify the domain of the
applications as a combination of the relevant parameters of the algorithm. For genetic
algorithms these might be the number of individuals of the population used by the optimizer
and the number of generations for which to run the optimizer. The following example in
pseudo-code demonstrates how to generate the tasks for case previously discussed.

```plaintext
individuals = {100, 200, 300, 500, 1000}
generations = {50, 100, 200, 400}
foreach indiv in individuals do
  foreach generation in generations do
    task = generate_task(indiv, generation)
    submit_task(task)
```

The algorithm sketched in the example defines a bi-dimensional domain composed by
discrete variables and then iterated over each combination of individuals and generations to
generate all the tasks composing the application. In this case 20 tasks are generated. The
function `generate_task` is specific to the application and creates the task instance by
substituting the values of `indiv` and `generation` to the corresponding variables in the template
definition. The function `submit_task` is specific to the middleware used and performs the
actual task submission.

A template task is in general a composition of operations concerning the execution of legacy
applications with the appropriate parameters and set of file system operations for moving
data. Therefore, frameworks natively supporting the execution of parameter sweep
applications often provide a set of useful commands that are used in to manipulate or to
operate with files. Also, the template task is often expressed as single file that composes
together the commands provided. The commonly available commands are:
- **Execute**: executes a program on the remote node.
- **Copy**: copies a file to/from the remote node.
- **Substitute**: substitute the parameter values with their placeholders inside a file.
- **Delete**: deletes a file.

All these commands can operate with parameters that are substituted with their actual values for each task instance.

Figure 7.2 and Figure 7.3 provide an example of two possible task templates. The former as defined according to the notation used by Nimrod/G and the latter as required by Aneka.

The template file has two sections: a header for the definition of the parameters and task definition section including shell commands mixed to Nimrod/G commands. The prefix `node:` identifies the remote location where the task is executed. Parameters are identified with the `{...}` notation. The example shown remotely executes the echo command and copies to the local user directory the output of the command by saving it into a file named according to the values of the parameters `x` and `y`.

The Aneka Parameter Sweep file defines the template task for executing the BLAST application. The file is an XML document containing several sections, the most important sections are `sharedFiles`, `parameters`, and `task`. The first contains the definition of the parameters that will customize the template task. Two different types of parameters are defined: a single value and a range parameter. The second section contains the files that are required to execute the task, while the third one specifies the operations that characterize the template task. The task has a collection of input and output files whose local and remote paths are defined, and a collection of commands. In the case presented a simple execute command is shown. With respect to the previous example there is no need to explicitly move the files to the remote destination but this operation is automatically performed by Aneka.
7.2.3 MPI Applications

MPI (Message Passing Interface) is a specification for developing parallel programs which communicate by exchanging messages. Compared to earlier models, MPI introduces the constraint of communication which involves that MPI tasks need to run at the same time. MPI has originated as an attempt to create a common ground from the several distributed shared memory and message passing infrastructures available for distributed computing. Nowadays, it has become a de-facto standard for developing portable and efficient message passing HPC applications. Interface specification have been defined and implemented for C/C++ and FORTRAN.

MPI provides developers with a set of routines that:

- Manage the distributed environment where MPI programs are executed.
- Provide facilities for point to point communication.
- Provide facilities for group communication.
- Provide support for data structure definition and memory allocation.
• Provide basic support for synchronization with blocking calls.

The general reference architecture is depicted in Figure 7.4. A distributed application in MPI is composed by a collection of MPI processes which are executed in parallel in a distributed infrastructure supporting MPI (most likely a cluster or nodes leased from Clouds).

![Figure 7.4. MPI Reference Scenario.](image)

MPI applications that share the same MPI runtime are by default part of a global group called `MPI_COMM_WORLD`. Within this group all the distributed processes have a unique identifier that allows the MPI runtime to localize and address them. It is possible to create specific groups as subsets of this global group for example for isolating all the MPI processes that belong to the same application. Each MPI process is assigned a rank within the group he belongs to. The rank is a unique identifier that allows processes to communicate with each other within a group. Communication is made possible by means of a communicator component that can be defined for each group.
In order to create an MPI application it is necessary to define the code for the MPI process that will be executed in parallel. This program has, in general, the structure described in Figure 7.5. The section of code that is executed in parallel is clearly identified by two operations that setup the MPI environment and shut it down respectively. In the code section defined within these two operations it is possible to use all the MPI functions to send or receive messages in either asynchronous or synchronous mode.

The diagram might suggest the MPI might allow the definition of completely symmetrical applications since the portion of code executed in each node is the same. In reality, it is possible to implement distributed applications based on complex communication patterns by differentiating the operations performed by each node according to the rank of the program, which is known at runtime. A common model used in MPI is the master-worker model, where one MPI process (usually the one with rank 0) coordinates the execution of others that perform the same task.

Once the program has been defined in one of the available MPI implementation, it is compiled with a modified version of the compiler for the language. This compiler introduces additional code in order to properly and to manage the MPI runtime. The output of the compilation process can be run as a distributed application by using a specific tool provided with the MPI implementation.

A general installation supporting the execution of the MPI application is composed by a cluster. In this scenario MPI is normally installed in the shared file system and an MPI daemon is started on each node of the cluster in order to coordinate the parallel execution of MPI applications. Once the environment is set up, it is possible to run parallel applications by
using the tools provided with the MPI implementation and specify several options such as the number of nodes to use to run the application.

At present there are several MPI implementations that can be leveraged to develop distributed applications, and the MPI specifications have currently reached version 2. One of the most popular MPI software environments (http://www.mcs.anl.gov/mpi/) is developed by the Argonne National Laboratory, USA. MPI has gained a lot of success as a parallel and distributed programming model for CPU intensive mathematical computations such as linear systems solvers, matrix computations, finite element computations, linear algebra, and numerical simulations.

7.2.4 Workflow Applications with Task Dependencies

Workflow applications are characterized by a collection of tasks that do exhibit dependencies among them. Such dependencies, which are mostly data dependencies (i.e. the output of one task is a prerequisite of another task), determine the way in which the applications are scheduled and also where they are scheduled. Concerns in this case are related to provide a feasible sequencing of tasks and how to optimize the placement of tasks so that the movement of data is minimized.

What is a Workflow?

The term workflow has a long tradition among the business community, where the term is used to describe a composition of services that all together accomplish a business process. As defined by the Workflow Management Coalition a workflow is the automation of a business process, in whole or part, during which documents, information, or tasks are passed from one participant (a resource; human or machine) to another for action, according to a set of procedural rules [64]. The concept of workflow as a structured execution of tasks, which have dependencies on each other, has demonstrated to be useful to express many scientific experiments, and gave birth to the idea of scientific workflow. Many scientific experiments are composed as a combination of problem solving components, which, connected in a particular order, define the specific nature of the experiment. When such experiments exhibit a natural parallelism and need to execute a large amount of operations, or deal with huge quantities of data, it makes sense to execute them on a distributed infrastructure. In the case of scientific workflows, the process is identified by an application to run, the elements that are passed among participants are mostly tasks and data, and the participants are mostly computing or storage nodes. The set of procedural rules are defined by a workflow definition scheme which guides the scheduling of the application. A scientific workflow generally involves data management, analysis, simulation and middleware supporting the execution of workflow.

A scientific workflow is generally expressed by a directed acyclic graph (DAG) defining the dependencies among tasks or operations. The nodes on the DAG represent the tasks to be executed in a workflow application, while the arcs connecting the nodes identify the
dependencies among tasks and the data paths that connect the tasks. The most common
dependency that is realized through a DAG is data dependency, which means that the
output files of a task (or some of them) constitute the input files of another task. This
dependency is represented as an arc originating from the node identifying the first task and
terminating in the node identifying the second task.

The DAG in Figure 7.6 describes a sample Montage workflow. Montage is a toolkit for
assembling images into mosaics; it has been specially designed to support astronomers in
composing the images taken from different telescopes, or points of views, into a coherent
image. The toolkit provides several applications for manipulating images and composing
them together; some of the applications perform background re-projection, perspective
transformation, and brightness and color correction. The workflow depicted in Figure 7.6
describes the general process for composing a mosaic, the label put on the right describe the
different tasks that have to be performed in order to compose a mosaic. In the case presented
in the diagram a mosaic is composed out of seven images. The entire process can take
advantage of a distributed infrastructure for its execution since there are several operations
that can be performed in parallel. For each of the image files the following process has to be

Figure 7.6. Sample Montage Workflow.

performed: image file transfer, re-projection, calculation of the difference, and common plane placement. Therefore, each of the images can be processed in parallel for these tasks. Here is where a distributed infrastructure helps in executing workflows.

There might be another reason for executing workflows on a distributed infrastructure: it might be convenient to move the computation on a specific node because of data locality issues. For example, if an operation needs to access specific resources, which are only available on a specific node, that operation cannot be performed elsewhere, while the rest of the operations might not have the same requirements. A scientific experiment might involve the use of several problem solving components that might require the use of specific instrumentation, in this case all the tasks having these constraints need to be executed where the instrumentation is available, thus creating a distributed execution of a process which is not parallel in principle.

**Workflow Technologies**

Business oriented computing workflows are defined as an composition of services and there exist specific languages and standards for the definition of workflows, such as BPEL (Business Process Execution Language) [65], in the case of scientific computing there is no common ground for defining workflows but several solutions and workflow languages coexist [66]. Despite such differences it is possible to identify an abstract reference model for a Workflow Management System [67], as depicted in Figure 7.7. Design tools allow users to visually compose a workflow application. This specification is normally stored in the form of an XML document, based on a specific workflow language, and constitutes the input of the workflow engine, which controls the execution of the workflow by leveraging a distributed infrastructure. In most of the cases, workflow engine is a client side component that might interact directly with resources or with one or several middleware for executing the workflow. Some frameworks can natively support the execution of workflow applications by providing a scheduler capable of directly processing the workflow specification.

Some of the most relevant technologies for designing and executing workflow-based applications are: Kepler, DAGMan, Cloudbus Workflow Management System, and Offspring.
Kepler [68] is an open source scientific workflow engine built from the collaboration of several research projects. The system is based on the Ptolemy II system [72], which provides a solid platform for developing dataflow-oriented workflows. Kepler provides a design environment based on the concept of actors, which are reusable and independent blocks of computation such as Web Services, database calls, etc. The connection between actors is made with ports. An actor consumes data from the input ports and writes data/results to the output ports. Novelty of Kepler is in its ability of separating the flow of data among components from the coordination logic that is used to execute workflow. Thus for the same workflow, it supports different models such as synchronous and asynchronous models. The workflow specification is expressed using a proprietary XML language.

DAGMan (Directed Acyclic Graph Manager) [69] is part of the Condor [5] project and constitutes an extension to the Condor scheduler to handle job inter-dependencies. Condor finds machines for the execution of programs, but does not support the scheduling of jobs in a specific sequence. Therefore, DAGMan acts as a meta-scheduler for Condor by submitting the jobs to the scheduler in the appropriate order. The input of DAGMan is a simple text file that contains the information about the jobs, pointers to their job submission files, and the dependencies among jobs.

Figure 7.7. Abstract Model of a Workflow System.
Cloudbus Workflow Management System (WfMS) [70] is a middleware platform built for managing large application workflows on distributed computing platforms such as Grids and Clouds. It comprises of software tools that help end-users to compose, schedule, execute, and monitor workflow applications through a web-based portal. The portal provides the capability of uploading workflows or defining new ones with a graphical editor. In order to execute workflows WfMS relies on the Gridbus Broker that is Grid/Cloud resource broker supporting the execution of applications with Quality of Service attributes over a heterogeneous distributed computing infrastructure including Linux based clusters, Globus, and Amazon EC2. WfMS uses a proprietary XML language for the specification of workflows.

A different perspective is taken by Offspring [71], which offers a programming based approach for developing workflows. Users can develop strategies and plug them into the environment, which will execute them by leveraging a specific distribution engine. The advantage provided by Offspring over other solutions is the ability of defining dynamic workflows. This strategy represents a semi-structured workflow that can change its behavior at runtime according to the execution of specific tasks. This allows developers to dynamically control the dependencies of tasks at runtime rather than statically defining them. Offspring supports the integration with any distributed computing middleware that can manage simple bag of tasks application. It provides a native integration with Aneka and also supports a simulated distribution engine for testing strategies during the development. As it allows the definition of workflows in the form of plug-ins, it does not use any XML specification.

7.3 Aneka Task-Based Programming

Aneka provides support for all the flavors of Task-based programming by means of the Task Programming Model, which constitutes the basic support given by the framework for supporting the execution of bag of tasks applications. Task programming is realized through the abstraction of the Aneka.Tasks.ITask. By using this abstraction as a basis support for execution of legacy applications, parameter sweep applications, and workflows have been integrated into the framework. In this section, we introduce the fundamental concepts of the model and provide examples on how to develop applications for all the previously discussed application models.

7.3.1 Task Programming Model

The task programming model provides a very intuitive abstraction for quickly developing distributed applications on top of Aneka. It provides a minimum set of APIs that are mostly centered on the Aneka.Tasks.ITask interface. This interface together with the services supporting the execution of tasks in the middleware constitutes the core feature of the model. Figure 7.8 provides an overall view of the components of the Task Programming Model and their role during application execution.
Developers create distributed applications in terms of \textit{ITask} instances whose collective execution describe a running application. These tasks, together with all the required dependencies (data files and libraries) are grouped together and managed through the \textit{AnekaApplication} class, which is specialized to support the execution of tasks. Two other components constitute the client side view of a task based application: \textit{AnekaTask} and \textit{TaskManager}. The former constitute the runtime wrapper used by Aneka to represent a task within the middleware, while the latter is the underlying component that interacts with Aneka, submits the tasks, monitor their execution, and collect the results. In the middleware, four services coordinate their activities in order to execute task based applications. These are: \textit{MembershipCatalogue}, \textit{TaskScheduler}, \textit{ExecutionService}, and \textit{StorageService}. \textit{MembershipCatalogue} constitutes the main access point of the Cloud and acts as service directory used to locate the \textit{TaskScheduler} service that is in charge of managing the execution of task based applications. Its main responsibility is to allocate task instances to resources featuring the \textit{ExecutionService} for task execution and monitoring task state. If the application requires the data transfer support in the form of data files, input or output files, an available \textit{StorageService} will be used as a staging facility for the application.

The features provided by the task model are completed by a Web Service that allows any client to submit the execution of tasks to Aneka. The procedure for submitting tasks through the web services is the same as the one done by using the framework APIs. The user creates an application on Aneka and submits tasks within the context of this application. The web service limits the type of tasks that can be submitted. Only a limited collection of tasks is available for submission; despite that, these tasks cover the functionality commonly found in other distributed computing systems.
Developing Applications with the Task Model

Execution of task-based applications involves several components. The development of such application is limited to the following operations:

- Defining classes implementing the `ITask` interface.
- Creating a properly configured `AnekaApplication` instance.
- Creating `ITask` instances and wrapping them into `AnekaTask` instances.
- Execute the application and wait for its completion.

Moreover, from a design point of view, the process of defining a task application ultimately reduces to the definition of the classes that implement `ITask`, which will be those that contribute to form the workload generated by the application.

**ITask and AnekaTask**

Almost all the client side features for developing task based application with Aneka are contained in the `Aneka.Tasks` namespace (`Aneka.Tasks.dll`). The most important component for designing tasks is the `ITask` interface, which is defined in Listing 1. This interface exposes only one method: `Execute`. The method is invoked in order to execute the task on the remote node.

```csharp
namespace Aneka.Tasks
{
    /// <summary>
    /// Interface ITask. Defines the interface for implementing a task.
    /// </summary>
    public interface ITask
    {
        /// <summary>
        /// Executes the sine function.
        /// </summary>
        ///
        public void Execute();
    }
}
```

The `ITask` interface provides a programming approach for developing native tasks, which means tasks implemented in any of the supported programming languages of .NET framework. The restrictions on implementing task classes are minimal, other than implementing the `ITask` interface, they need to be serializable since task instances are created and moved over the network. Listing 2 describes a simple implementation of a task class that computes the Gaussian distribution for a given point x.

```csharp
// File: GaussTask.cs
using System;
using Aneka.Tasks;

// Listing 2. ITask Interface Implementation.
```
namespace GaussSample
{
    /// <summary>
    /// Class GaussTask. Implements the ITask interface for computing the Gauss function.
    /// </summary>
    [Serializable]
    public class GaussTask : ITask
    {
        /// <summary>
        /// Input value.
        /// </summary>
        private double x;
        /// <summary>
        /// Gets the input value of the Gauss function.
        /// </summary>
        public double X { get { return this.x; } set { this.x = value; } }
        /// <summary>
        /// Result value.
        /// </summary>
        private double y;
        /// <summary>
        /// Gets the result value of the Gauss function.
        /// </summary>
        public double Y { get { return this.y; } set { this.y = value; } }
        /// <summary>
        /// Executes the Gauss function.
        /// </summary>
        public void Execute()
        {
            this.y = Math.Exp(-this.x * this.x);
        }
    }
}

ITask provides minimum restrictions on how to implement a task class and decouples the specific operation of the task from the runtime wrapper classes. It is required for managing tasks within Aneka. This role is performed by the AnekaTask class that represents the task instance in accordance with the Aneka application model APIs. This class extends the Aneka.Entity.WorkUnit class and provides the feature for embedding ITask instances. AnekaTask is mostly used internally and for end users it provides facilities for specifying input and output files for the task.

Listing 3. Wrapping an ITask into an AnekaTask instance.

// create a Gauss task and wraps it into an AnekaTask instance
GaussTask gauss = new GaussTask();
AnekaTask task = new AnekaTask(gauss);
// add one input and one output files
task.AddFile("input.txt", FileDataType.Input, FileAttributes.Local);
task.AddFile("result.txt", FileDataType.Output, FileAttributes.Local);

Listing 3 describes how to wrap an ITask instance into an AnekaTask. It also shows how to add input and output files specific to a given task. The Task Programming Model leverages the basic capabilities for file management that belong to the WorkUnit class, which the
**AnekaTask** class inherits from. As discussed while presenting the Aneka Application Model (see Chapter 5 —) **WorkUnit** has two collection of files **InputFiles** and **OutputFiles**; developers can add files to these collections and the runtime environment will automatically move these files where it is necessary. Input files will be staged into the Aneka Cloud and moved to the remote node where the task is executed. Output files will be collected from the execution node and moved to the local machine or a remote FTP server.

### Controlling Task Execution

Task classes and **AnekaTask** define the computation logic of a task based application, while the **AnekaApplication** class provides the basic feature for implementing the coordination logic of the application.

**AnekaApplication** is a generic class that can be specialized for supporting different programming models, in the case of the *Task Programming* it assumes the form of **AnekaApplication<AnekaTask, TaskManager>**. For other programming models the class provides the following operations:

- Static and dynamic task submission.
- Application state and task state monitoring.
- Event-based notification of tasks completion or failure.

By composing these features all together it is possible to define the logic that is required to implement a specific task application. Static submission is a very common pattern in the case of task-based applications and it involves the creation of all the tasks that need to be executed in one loop and their submission as a single bag. More complex task submission strategies are then required for implementing workflow based applications, where the execution of tasks is determined by dependencies among them. In this case a dynamic submission of tasks is a more efficient technique and involves the submission of tasks as a result of the event-based notification mechanism implemented in the **AnekaApplication** class.

---

#### Listing 4. Static Task Submission.

```java
// get an instance of the Configuration class from file
Configuration conf = Configuration.GetConfiguration("conf.xml");
// specify that the submission of task is static (all at once)
cnf.SingleSubmission = true;
AnekaApplication<AnekaTask, TaskManager> app =
new AnekaApplication<Task,TaskManager>(conf);
for(int i=0; i<400; i++)
{
    GaussTask gauss = new GaussTask();
    gauss.X = i;
    AnekaTask task = new AnekaTask(gauss);
    // add the task to the bag of work units to submit
    app.AddWorkunit(task);
}
// submit the entire bag
app.SubmitExecution();
```

---

257
Listing 4 shows how to create and submit 400 Gauss tasks as a bag by using the static submission approach. The *AnekaApplication* class has a collection of tasks containing all the tasks that have been submitted for execution, each task can be referenced by using its unique identifier (*WorkUnit.Id*) by the indexer operator [] applied to the application class. In the case of static submission the tasks are added to the application and the method *SubmitExecution()* is called.

A different scenario is constituted by dynamic submission, where tasks are submitted as a result of other events occurring during the execution, for example the completion or the failure of previously submitted tasks or other conditions that are not related with the interaction of Aneka. In this case, developers have more freedom in selecting the most appropriate task submission strategy. For example, it is possible that an initial bag of tasks is submitted as described in the previous listing, and subsequently as a result of the completion of some tasks other tasks are generated and submitted. In order to implement this scenario it is necessary to rely on the event-based notification system provided by the *AnekaApplication* class and trigger the submission of tasks according to the firing of specific events. In particular, we are interested in the *WorkUnitFailed* and *WorkUnitCompleted* events.

```
/// <summary>
/// Main method for submitting tasks.
/// </summary>
public void SubmitApplication()
{
    // get an instance of the Configuration class from file
    Configuration conf = Configuration.GetConfiguration("conf.xml");
    // specify that the submission of task is dynamic
    conf.SingleSubmission = false;
    AnekaApplication<AnekaTask, TaskManager> app = new AnekaApplication<AnekaTask, TaskManager>(conf);
    // attach methods to the event handler that notify the client code
    // when tasks are completed or failed
    app.WorkUnitFailed +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
    app.WorkUnitFinished +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);
    for(int i=0; i<400; i++)
    {
        GaussTask gauss = new GaussTask();
        gauss.X = i;
        AnekaTask task = new AnekaTask(gauss);
        // add the task to the bag of work units to submit
        app.AddWorkunit(task);
    }
    // submit the entire bag
    app.SubmitExecution();
}
```

Listing 5. Dynamic Task Submission.
Listing 5 extends the previous example and implements a dynamic task submission strategy for refining the computation of Gaussian distribution. Both static and dynamic task submissions are used: an initial bag of 400 Gauss task is submitted to Aneka and as soon as these tasks are completed a new task for the computation of an intermediate value of the distribution is submitted. In order to capture the failure and the completion of tasks it is necessary to listen to the events `WorkUnitFailed` and `WorkUnitFinished`. The event signature requires the methods to have an object parameter (as for all the event handlers), which will contain the application instance, and a `WorkUnitEventArgs<AnekaTask>` argument containing the information about the `WorkUnit` that triggered the event. This class exposes a `WorkUnit` property that, if not null, gives access to the task instance. The event handler for the task failure simply dumps the information that the task is failed to the console, with additional information about the error occurred if possible. The event handler for task completion checks whether the task completed was submitted within the original bag and in this case submits another task by using the `ExecuteWorkUnit(AnekaTask task)` method. In order to
discriminate tasks submitted within the initial bag and other tasks the value of GaussTask.X is used. If X contains a value with no fractional digits, it is an initial task, otherwise not.

Static and dynamic submission influence the way in which the application termination condition is determined. In case of static submission the determination of this condition is automatic: once all the tasks initially submitted are failed or completed the application is terminated. It is important in this case to activate in the configuration of the application the SingleSubmission flag by setting it to true. This will tell the runtime to automatically determine the completion of the application. In case of dynamic submission, it is impossible for the runtime to determine the termination of the application since it is always possible to submit new tasks. In this case it is responsibility of the developer to signal the application class the termination of the application by invoking StopExecution method when appropriate.

While designing the coordination logic of the application, it is important to note that the task submission identifies an asynchronous execution pattern, which means that the method SubmitExecution, as well as the method ExecuteWorkUnit, returns when the submission of tasks is completed, but not the actual completion of tasks. This requires the developer to put in place the proper synchronization logic in order to let the main thread of the application to wait until all the task are terminated and the application is completed. This behaviour can be implemented by using the synchronization APIs provided by the System.Threading namespace: System.Threading.AutoResetEvent or System.Threading.ManualResetEvent. These two, together with a minimal logic count all the tasks to be collected and signal the main thread (put in waiting state by calling the method WaitHandle.Wait()) once all tasks are terminated. It can also provide the required infrastructure for properly managing the execution flow of the application. Listing 6 provides a complete implementation of the task submission program implementing dynamic submission and the appropriate synchronization logic.


```csharp
// File: GaussApp.cs
using System;
using System.Threading;
using Aneka.Entity;
using Aneka.Tasks;

namespace GaussSample
{
    /// <summary>
    /// Class GaussApp. Defines the coordination logic of the
    /// distributed application for computing the gaussian distribution.
    /// </summary>
    public class GaussApp
    {
        /// <summary>
        /// Semaphore used to make the main thread wait while
        /// all the tasks are terminated.
        /// </summary>
        private ManualResetEvent semaphore;
    }
}
```
/// <summary>
/// Counter of the running tasks.
/// </summary>
private int taskCount = 0;

/// <summary>
/// Aneka application instance.
/// </summary>
private AnekaApplication<AnekaTask, TaskManager> app;

/// <summary>
/// Main entry point for the application.
/// </summary>
/// <param name="args">An array of strings containing the command line.</param>
public static void Main(string[] args)
{
    try
    {
        // initialize the logging system
        Logger.Start();

        string confFile = "conf.xml";
        if (args.Length > 0)
        {
            confFile = args[0];
        }
        // get an instance of the Configuration class from file
        Configuration conf = Configuration.GetConfiguration(confFile);
        // create an instance of the GaussApp and starts its execution
        // with the given configuration instance
        GaussApp application = new GaussApp();
        application.SubmitApplication(conf);
    }
    catch (Exception ex)
    {
        IOUtil.DumpErrorReport(ex, "Fatal error while executing application.");
    }
    finally
    {
        // terminate the logging thread
        Logger.Stop();
    }
}

/// <summary>
/// Application submission method.
/// </summary>
/// <param name="conf">Application configuration.</param>
public void SubmitApplication(Configuration conf)
{
    // initialize the semaphore and the number of
    // task initially submitted
    this.semaphore = new ManualResetEvent(false);
    this.taskCount = 400;

    // specify that the submission of task is dynamic
    conf.SingleSubmission = false;
    this.app = new AnekaApplication<Task, TaskManager>(conf);
    // attach methods to the event handler that notify the client code
    // when tasks are completed or failed
    this.app.WorkUnitFailed +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
    this.app.WorkUnitFinished +=
        new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFailed);
}
new EventHandler<WorkUnitEventArgs<AnekaTask>>(this.OnWorkUnitFinished);

// attach the method OnAppFinished to the Finished event so we can capture
// the application termination condition, this event will be fired in case of
// both static application submission or dynamic application submission
app.Finished += new EventHandler<ApplicationEventArgs>(this.OnAppFinished);

for(int i=0; i<400; i++)
{
    GaussTask gauss = new GaussTask();
    gauss.X = i;
    AnekaTask task = new AnekaTask(gauss);
    // add the task to the bag of work units to submit
    app.AddWorkunit(task);
}
// submit the entire bag
app.SubmitExecution();

// wait until signaled, once the thread is signaled the application is completed
this.semaphore.Wait();

/// <summary>
/// Event handler for task failure.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFailed(object sender, WorkUnitEventArgs<AnekaTask> args)
{
    // do nothing, we are not interested in task failure at the moment
    // just dump to console the failure.
    if (args.WorkUnit != null)
    {
        Exception error = args.WorkUnit.Exception;
        Console.WriteLine("Task {0} failed - Exception: {1}",
            args.WorkUnit.Name,
            (error == null ? "[Not given]" : error.Message);
    }
    // we do not have to synchronize this operation because
    // events handlers are run all in the same thread, and there
    // will not be other threads updating this variable
    this.taskCount--;
    if (this.taskCount == 0)
    {
        this.app.StopExecution();
    }
}

/// <summary>
/// Event handler for task completion.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, WorkUnitEventArgs<AnekaTask> args)
{
    // do nothing, we are not interested in task failure at the moment
    // just dump to console the failure.
    if (args.WorkUnit != null)
    {
        GaussTask gauss = (GaussTask) args.WorkUnit.Task;
        // we check whether it is an initially submitted task or a task
// that we submitted as a reaction to the completion of another task
if (task.X - Math.Abs(task.X) == 0)
{
    // ok it was an original task, then we increment of 0.5 the
    // value of X and submit another task
    GaussTask fraq = GaussTask();
    fraq.X = gauss.X + 0.5;
    AnekaTask task = new AnekaTask(fraq);
    this.taskCount++;
    // we call the ExecuteWorkUnit method that is used
    // for dynamic submission
    app.ExecuteWorkUnit(task);
}
Console.WriteLine("Task {0} completed – [X:{1},Y:{2}]",
    args.WorkUnit.Name, gauss.X, gauss.Y);
if (this.taskCount == 0)
{
    this.app.StopExecution();
}
}
/// <summary>
/// Event handler for the application termination.
/// </summary>
/// <param name="sender">Event source: the application instance.</param>
/// <param name="args">Event arguments.</param>
private void OnWorkUnitFinished(object sender, ApplicationEventArgs args)
{
    // unblock the main thread, because we have identified the termination
    // of the application
    this.semaphore.Set();
}
}

The listing provides a complete definition of the GaussApp class, which also contain the main entry point of the application. A very simple logic for controlling the execution of the application has been implemented. The GaussApp application keeps track of the number of the currently running tasks by using the taskCount field, when this value reaches zero there are no more tasks to wait for and the application is stopped by calling StopExecution. This method fires the ApplicationFinished event whose event handler (OnApplicationFinished) unblocks the main thread by signalling the semaphore. The value of taskCount is initially set to 400, which is the size of the initial bag of tasks. Every time a task fails or completes this field is decremented by one unit and in case there is a new task submission the field is incremented by one unit. At the end of the two event handlers (OnWorkUnitFailed and OnWorkUnitFinished) the value of taskCount is checked to see whether it is equal to zero and it is necessary to stop the application. We can observe that, besides the use of the ManualResetEvent, there is no need of other synchronization structures. As the code that manipulates the value of taskCount is executed in one single thread, there will not be any races while incrementing or decrementing the value.

A final aspect that can be considered for controlling the execution of the task application is the resubmission strategy that is used. By default the configuration of the application sets the
resubmission strategy as manual; this means that if a task fails because of an exception occurred during its execution, the task instance is sent back to the client application and it is responsibility of the developer to resubmit the task if necessary. In case of automatic resubmission, Aneka will keep resubmitting the task until a maximum number of attempts is reached. If the task keeps failing, then the task failure event will be eventually fired. This property can be controlled by setting the configuration value Configuration.ResubmitMode to ResubmitMode.Manual or ResubmitMode.Auto. As previously, said this property is set to ResubmitMode.Manual by default.

File Management

Task based application normally deal with files in order to perform their operations. As already discusses files may constitute input data for tasks, may contain the result of a computation, or may represent executable code or library dependencies. Therefore, providing support for file transfer for task based application is essential. Aneka provides built-in capabilities for file management in a distributed infrastructure and the Task Programming Model transparently leverages these capabilities. Any model based on the WorkUnit and the ApplicationBase classes has built-in support for file management. It is possible to provide input files that are common to all the WorkUnit instances, through the ApplicationBase.SharedFiles collection, and instance specific input and output files by leveraging the WorkUnit.InputFiles and WorkUnit.OutputFiles collections.

A fundamental component for the management of files is the FileData class, which constitutes the logic representation of physical file, defined in the Aneka.Data.Entity namespace (Aneka.Data.dll). A FileData instance provides information about a file:

- Its nature: whether it is a shared file, an input file, or an output file.
- Its path both in the local and in the remote file system including a different name.
- A collection of attributes that provides other information (such as the final destination of the file or whether the file is transient or not, etc.).

By using the FileData class the user specifies the file dependencies of tasks and the application and the Aneka APIs will automatically transfer them to and from the Cloud when needed. Aneka allows specifying both local and remote files stored on FTP servers or Amazon S3. A FileData instance is identified by three elements: an owner, a name, and a type. The owner identifies, by means of the corresponding id, which is the computing element that needs the file: application instance or work unit. The type specifies whether file is shared, input or output file. The name represents the name of the corresponding physical file.

Listing 7. File Dependencies Management.

```csharp
// get an instance of the Configuration class from file
Configuration conf = Configuration.GetConfiguration("conf.xml");
AnekaApplication<Task,TaskManager>app =new AnekaApplication<Task,TaskManager>(conf);

// attach shared files with different methods by using the FileData class and directly
```
// using the API provided by the AnekaApplication class

// create a local shared file whose local and remote name is "pi.tab"
FileDatapiTab = newFileData("pi.tab", FileDataType.Shared);
app.AddSharedFile(piTab);
// once the file is added to the collection of shared files, its OwnerId property
// references app.Id

// create a remote shared file by specifying the attributes whose name is "pi.dat"
FileDatapiDat = newFileData("pi.dat", FileDataType.Shared, FileAttributes.None);
// the StorageBucketId property points a specific configuration section that is
// used to store the information for retrieving the file from the remote server
piDat.StorageBucketId = "FTPStore";
app.AddSharedFile(piDat);
// once the file is added to the collection of shared files, its OwnerId property
// references app.Id

// adds a local shared file
app.AddSharedFile("pi.xml");

for(int i=0; i<400; i++)
{
    GaussTask gauss = new GaussTask();
gauss.X = i;
AnekaTask task = new AnekaTask(gauss);

    // adds a local input file for the current task whose name is "<i>.txt"
    // where <i> is the value of the loop variable
    FileData input = new FileData(string.Format("{0}.txt", i);
    FileDataType.Input, FileAttributes.Local);
    // once transferred to the remote node, the file will have the name
    // "input.txt". Since tasks are executed in separate directories there
    // will be no name clashing
    input.VirtualPath = "input.txt";
    task.AddFile(input);
    // once the file is added to the task, it will be stored in the InputFiles
    // collection and its OwnerId property will referenced task.Id

    // adds an output file for the current task whose name is "out.txt" that will
    // be stored on S3
    FileData output = new FileData("out.txt", FileDataType.Output, FileAttributes.None);
    // once transferred to the remote server, the file will have the name
    // "<i>.out" where <i> is the value of the loop variable. In this way we
    // easily avoid name clashing while storing output files into a single
    // directory
    output.VirtualPath = string.Format("{0}.out", i);
    output.StorageBucketId = "S3Store";
    task.AddFile(output);
    // once the file is added to the task, it will be stored in the InputFiles
    // collection and its OwnerId property will referenced task.Id

    // adds a local output file for the current task whose name is "trace.log".
    // The file is optional, this means that if after the execution of the task the file
    // is not present no exception or task failure will be risen.
    FileData trace = new FileData("trace.log", FileDataType.Output,
        FileAttributes.Local | FileAttributes.Optional);
    // once transferred to the local machine, the file will have the name
    // "<i>.log" where <i> is the value of the loop variable. In this way we
    // easily avoid name clashing while storing output files into a single
    // directory
    trace.VirtualPath = string.Format("{0}.log", i);
    task.AddFile(trace);
Listing 7 demonstrates how to add file dependencies to the application and to tasks. It is possible to add both FileData instances, thus having more control on the information attached to the file, or to use more intuitive approaches that just require the name and the type of the file.

The general interaction flow for file management is the following:

- Once the application is submitted the shared files are staged into the Aneka Cloud.
- If the file is local it will be searched into the directory pointed by Configuration.Workspace, if the file is remote the specific configuration settings mapped by the FileData.StorageBucketId property will be used to access the remote server and stage in the file.
- If there is any failure in staging input files, the application is terminated with an error.
- For each of the tasks belonging to the application the corresponding input files are staged into the Aneka Cloud as done for shared files.
- Once the task is dispatched for execution to a remote node, the runtime will transfer all the shared files of the application and the input files of the task into the working directory of the task and eventually get renamed if the FileData.VirtualPath property is not null.
- After the execution of the task the runtime will look for all the files that have been added into the WorkUnit.OutputFiles collection. If not null, the value of the FileData.VirtualPath property will be used to locate the files, otherwise the FileData.FileName property will be the reference. All the files that do not contain the FileAttributes.Optional attribute need to be present; otherwise the execution of the task is classified as a failure.
- Despite the successful execution or the failure of a task, the runtime tries to collect and move to their respective destinations all the files that are found. Files that contain the FileAttributes.Local attributed are moved to the local machine from where the application is saved and stored in the directory pointed by Configuration.Workspace. Files that have a StorageBucketId property set will be staged out to the corresponding remote server.

The infrastructure for file management provides a transparent and extensible service for the movement of data. The architecture for the management of files is based on the concept of factories and storage buckets. A factory, namely IFileTransferFactory, is a component that is used to abstract the creation of client and server component for file transfer, so that the entire
architecture can work with interfaces rather than specific implementations. Storage buckets are a collection of string properties that are used to specialise these components through configuration files. Storage buckets are specified by the user, either by the configuration file or by programmatically adding this information to the configuration object, before submitting the application. Factories are used by the StorageService to pull in remote input and shared files and to pull out remote output files.


```xml
<?xml version="1.0" encoding="utf-8"?>
<Aneka>
  <UseFileTransfer value="true" />
  <Workspace value="." />
  <SingleSubmission value="false" />
  <ResubmitMode value="Manual" />
  <PollingTime value="1000" />
  <LogMessages value="true" />
  <SchedulerUri value="tcp://localhost:9090/Aneka"/>
    <UserCredentials username="Administrator" password=""/>
  </UserCredential>
  <Groups>
    <Group name="StorageBuckets">
      <Group name="FTPStore">
        <Property name="Scheme" value="ftp"/>
        <Property name="Host" value="www.remoteftp.org"/>
        <Property name="Port" value="21"/>
        <Property name="Username" value="anonymous"/>
        <Property name="Password" value="nil"/>
      </Group>
      <Group name="S3Store">
        <Property name="Scheme" value="S3"/>
        <Property name="Host" value="www.remoteftp.org"/>
        <Property name="Port" value="21"/>
        <Property name="Username" value="anonymous"/>
        <Property name="Password" value="nil"/>
      </Group>
    </Group>
  </Groups>
</Aneka>
```

Listing 8 shows a sample configuration file containing the settings required to access the remote files through the FTP and S3 protocols. Within the <Groups> tag, there is a specific group whose name is StorageBuckets; this group maintains the configuration settings for each storage bucket that needs to be used in for file transfer. Each <Group> tag represents a storage bucket and the name property contains the values referenced in the FileData.StorageBucketId property. The content of each of these groups is specific to the type of storage bucket used, which is identified by the Scheme property. These values are used by the specific implementations for the FTP and S3 protocols to access the remote servers and transfer the files.
Task Libraries

Aneka provides a set of ready to use tasks for performing the most basic operations for remote file management. These tasks are part of the Aneka.Tasks.BaseTasks namespace, which is part of the Aneka.Tasks.dll library. The following operations are implemented:

- **File copy.** The `LocalCopyTask` performs the copy of a file on the remote node; it takes as input a file and produces a copy of it under a different name or path.
- **Legacy application execution.** The `ExecuteTask` allows executing external and legacy applications by using the `System.Diagnostics.Process` class. It requires the location of the executable file to run and it is also possible to specify command line parameters. `ExecuteTask` also collect the standard error and standard output produced by the execution of the application.
- **Substitute operation.** The `SubstituteTask` performs a search and replace operation within a given file by saving the resulting file under a different name. It is possible to specify a collection of string-based name-value pairs representing the strings to search together with their corresponding replacements.
- **File deletion.** The `DeleteTask` deletes a file that is accessible through the file system on the remote node.
- **Timed delay.** The `WaitTask` introduces a timed delay. This task can be used in several scenarios, for example it can be used for profiling or simply simulation of the execution. Also, it can also be used to introduce a pause between the executions of two applications if needed.
- **Task composition.** The `CompositeTask` implements the composite pattern and allows expressing a task as a composition of multiple tasks that are executed in sequence. This task is very useful to perform complex tasks involving the combination of operations implemented in other tasks.

The base task library does not provide any support for data transfer since this operation is automatically performed by the infrastructure when needed. Beside these simple tasks, the Aneka API allows for the creation of any user defined task by simply implementing the `ITask` interface and supporting object serialization.

Web Services Integration

Aneka provides integration with other technologies and applications by means of Web Services, which allow some of the services hosted in the Cloud to be accessible in platform

---

40 In software engineering, the composite pattern is a software design pattern, which allows expressing a combination of components as a single component. The advantage of using such pattern resides in creating a software infrastructure that allows forwarding the execution of an operation to a group of objects by treating it as single unit and in completely transparent manner. Reference: E. Gamma, R. Helm, R. Johnson, and J.M. Vlissides, Design Patterns: Elements of Reusable Software Design, Addison Wesley, 1995, ISBN: 0201633612.
independent fashion. Among these, the task submission web service allows third party applications to submit tasks as it happens in traditional Computing Grids.

![Diagram: Web Service Submission Scenario](image)

The task submission web service is an additional component that can be deployed in any ASP.NET web server and that exposes a simple interface for job submission, which is compliant with the Aneka application model. The task web service provides an interface that is more compliant with the traditional way fostered by Grid Computing. Therefore, the new concept of job, which is a collection of predefined tasks, is introduced. The reference scenario for web based submission is depicted in Figure 7.9. Users create a distributed application instance on the Cloud and within the context of this application they can submit jobs querying the status of the application or a single job. It is up to the users to then terminate the application when all the jobs are completed or to abort it if there is no need to complete their execution.

Jobs can be composed by putting together the tasks defined in the basic task library. Operations supported through the web service interface are the following:

- Local file copy on the remote node.
- File deletion.
- Legacy application execution through the common shell services.
- Parameter substitution.

It is also possible specify input and output files for each job. The only restriction in this case is that both input and output files need to reside in remote FTP servers. This enables Aneka to automatically stage the files from these servers without the user intervention. Figure 7.10
gives detailed information about the object model exposed through the web service for job submission.

Traditional grid technologies such as the Gridbus Broker [15] and the Workflow Engine [70] can make use of task web service to submit their tasks for execution on Cloud nodes managed by Aneka.

### 7.3.3 Developing Parameter Sweep Application

Aneka integrates support for parameter sweeping applications on top of the **Task Model** by means of a collection of client components allowing developers to quickly prototype applications either through programming APIs or graphical user interfaces. The set of abstractions and tools supporting the development of parameter sweep applications constitutes the **Parameter Sweep Model (PSM)**.

The model is organized into several namespaces under the common root `Aneka.PSM`. More precisely:

- **Aneka.PSM.Core** (`Aneka.PSM.Core.dll`) contains the base classes for defining a template task and the client components managing the generation of tasks given the set of parameters.
- **Aneka.PSM.Workbench** (`Aneka.PSM.Workbench.exe`) and **Aneka.PSM.Wizard** (`Aneka.PSM.Wizard.dll`) contain the user interface support for designing and monitoring parameter sweep applications. Mostly they contain the classes and components...
required by the Design Explorer, which is the main GUI for developing parameter sweep applications.

- Aneka.PSM.Console (Aneka.PSM.Console.exe) contains the components and classes supporting the execution of parameter sweep applications in console mode.

These namespace define the support for developing and controlling parameter sweep application on top of Aneka.

Object Model

The fundamental elements of the Parameter Sweep Model are defined in the Aneka.PSM.Core namespace. This model introduces the concept of job (Aneka.PSM.Core.PSMJobInfo), which identifies a parameter sweep application. A job comprises file dependencies, parameter definitions together with their admissible domain, and the definition of the template task. Figure 7.11 shows the most relevant component of the object model.

The root component for application design is the PSMJobInfo class, which contains information about shared files, input and output files (PSMFileInfo). In accordance with the Aneka application model, shared files are common to all the instances of the template task, while input and output files are specific to a given task instance. Therefore, these ones can be expressed as a function of the parameters. Currently, it is possible to specify five different types of parameters:

- **Constant parameter** (PSMSingleParameterInfo): this parameter identifies a specific value that is set at design time and will not change during the execution of the application.
- **Range parameter** (PSMRangeParameterInfo): this parameter allows defining a range of allowed values, which might be integer or real. The parameter identifies a domain composed by discrete values and requires the specification of a lower bound, an upper bound, and a step for the generation of all the admissible values.
- **Random parameter** (PSMRandomParameterInfo): this parameter allows the generation of a random value in between a given range defined by a lower and an upper bound. The value generated is real.
- **Enumeration parameter** (PSMEnumParameterInfo): this parameter allows specifying a discrete set of values of any type. It is useful to specify discrete sets that are not based on numeric values.
- **System parameter** (PSMSystemParameterInfo): this parameter allows mapping a specific value that will be substituted at runtime while the task instance is executed on the remote node.
Other than these parameters, the object model reserves special parameters that are used to identify specific values of the PSM object model, such as the task identifier and other data. Parameters have access to the execution environment by means of an execution context (PSMContext) that is responsible for providing default and runtime values. The task template is defined as a collection of commands (PSMCommandInfo), which replicate and extend the...
features available in the base task library. The available commands for composing the task template perform the following operations:

- Local file copy on the remote node (`PSMCopyCommandInfo`).
- Remote file deletion (`PSMDeleteCommandInfo`).
- Execution of programs through the shell (`PSMExecuteCommandInfo`).
- Environment variable setting on the remote node (`PSMEnvironmentCommandInfo`).
- String pattern replacement within files (`PSMSubstituteCommandInfo`).

By following the same approach described for the creation of tasks, it is possible to define the task template by composing these basic blocks. All the properties exposed by these commands can include the previously defined parameters, whose values will be provided during the generation of the task instances.

![Diagram of Parameter Sweep Model APIs](image)

**Figure 7.12. Parameter Sweep Model APIs.**

A parameter sweep application is executed by means of a job manager (`IJobManager`), which interfaces the developer with the underlying APIs of the Task Model. Figure 7.12 shows the relation between the PSM APIs, with a specific reference to the job manager, and the Task Model APIs.

Through the `IJobManager` interface it is possible to specify user credentials and configuration for interacting with the Aneka middleware. The implementation of `IJobManager` will then create a corresponding Aneka application instance and leverage the Task Model API to submit all the task instances generated from the template task. The interface also exposes
facilities for controlling and monitoring the execution of the parameter sweep application, as well as support for registering the statistics about the application.

**Development and Monitoring Tools**

The core libraries allow developer to directly program parameter sweep applications and embed them into other applications. Additional tools simplify design and development of parameter sweep applications by providing support for visual design of the applications and interactive and non-interactive application execution. These tools are the *Aneka Design Explorer* and the *Aneka PSM Console*.

The *Aneka Design Explorer* is an integrated visual environment for quickly prototyping parameter sweep applications, executing them, and monitoring their status. It provides a simple wizard helping user to visually define any aspect of parameter sweep applications such as file dependencies and result files, parameters, and template tasks. The environment also provides a collection of components that help users to monitor application execution, aggregate statistics about application execution, detailed task transition monitoring, and extensive access to application logs.

The *Aneka PSM Console* is a command line utility designed to run parameter sweep applications in non-interactive mode. The console offers a simplified interface for running applications with the essential features for monitoring their execution. With respect to the *Design Explorer*, the console offers less support for keeping and visualizing aggregate statistics, but exposes the same data in a more simplified textual form.

### 7.3.4 Managing Workflows

Support for workflow in Aneka is not native, but obtained with plug-ins that allows client-based workflow managers to submit tasks to Aneka. Currently, two different workflow managers can leverage Aneka for task execution: the *Workflow Engine* and *Offspring*. The former leverages the task submission web service exposed by Aneka while the latter directly interacts with the Aneka programming APIs. The *Workflow Engine* plug-in for Aneka constitutes an example of the integration capabilities offered by the framework, which allows client applications developed with any technology and language to leverage Aneka for task execution. The integration developed for *Offspring* constitutes another example on how it is possible to construct another programming model on top of the existing APIs available in the framework. Therefore, we discuss this solution in more detail.

Figure 7.13 describes the architecture of *Offspring*. The system is composed by two types of components: plug-in and distribution engine. The former is used to enrich the environment of features while the latter represent the access to the distributed computing infrastructure leveraged for task execution. Among the available plug-ins, the *AutoPlugin* provides facilities for the definition of workflows in terms of *strategies*. A strategy generates the tasks that are submitted for execution and also defines the logic, in terms of sequencing, coordination, and
dependencies, used to submit the task through the engine. A specific component, the `StrategyController`, decouples the strategies from the distribution engine; therefore, strategies can be defined independently from the specific middleware used. The connection with Aneka is realized through the `AnekaEngine`, which implements the operations of `IDistributionEngine` for the Aneka middleware and relies on the services exposed by the `Task Model` programming APIs.

The systems allows for the execution of dynamic workflows, whose structure is defined as the workflow executes. Two different types of tasks can be defined: native tasks and legacy tasks. Native tasks are completely implemented in managed code. While legacy tasks manage file dependencies and wrap all the data necessary for the execution of legacy programs on a remote node. Also, a strategy may define shared file dependencies that are necessary to all the tasks generated by the workflow. The dependencies among tasks are implicitly defined by the execution of the strategy by the `StrategyController` and the events fired by the distributed engine.

Figure 7.13. Offspring Architecture.

Figure 7.14 describes the interactions among these components. Two main execution threads control the execution of a strategy. A `control thread` manages the execution of the strategy whereas a `monitoring thread` collects the feedback from the distribution engine and allows for the dynamic reaction of the strategy to the execution of tasks previously submitted. From the workflow developer’s point of view, the logic is quite simple. The execution of a strategy is composed by three macro steps: setup, execution, and finalization. The first step involves the
setup of the strategy and the application mapping it. Correspondingly, the finalization step is in charge of releasing all the internal resources allocated by the strategy and shutting down the application. The core of the workflow execution resides in the execution step, which is broken down into a set of iterations. During each of the iterations a collection of tasks is submitted, these tasks do not have dependencies from each other and can be executed in parallel. As soon as a task complete or fails the strategy is queried in order to see whether a new set of tasks need to be executed. In this way dependencies among tasks are implemented. If there are more tasks to be executed, they are submitted and the controller waits for a feedback from the engine, otherwise an iteration of the strategy is completed. At the end of each iteration, the controller checks whether the strategy has completed the execution and in this case, the finalization step is performed.

The \textit{AnekaEngine} creates an instance of the \textit{AnekaApplication} class for each execution of a strategy and configures the template class with a specific implementation of the \textit{TaskManager}, which overrides the behaviour implemented for file management and optimizes the staging of output files. In order to support the implementation of workflows without any dependency from the distribution engine, the application configuration settings are controlled by the distribution engine and shared among all the strategies executed through the engine.
7.4 Summary

In this chapter, we have introduced the task based programming and provided an overview of the technologies supporting the development of distributed applications based on the concept of tasks. Task based programming constitutes the most intuitive approach for distributing the computation of an application over a set of nodes. The main abstraction of task based programming is the concept of task, which represents a group of operations that can be isolated and executed as a single unit. A task can be a simple program that is executed through the shell or a more complex piece of code requiring a specific runtime environment to execute. Quite often, tasks require input files for their execution and produce output files as a result. According to this model, an application is expressed as a collection of tasks; the way in which these tasks are interrelated and their specific nature and characteristics differentiate the different models that are an expression of task based programming.

Traditionally, task based programming model has been successfully used in the development of distributed applications in many areas. We identified three major computing categories where task model can be utilized. High Performance Computing (HPC) refers to the use of distributed computing facilities for solving problems needing large computing power. Common HPC applications feature a large collection of compute intensive tasks whose duration is relatively short. High Throughput Computing (HTC) identifies scenarios where distributed computing facilities are used to support the execution of applications needing large computing power for a long period of time. Tasks may not be numerous, but have a long duration and infrastructure reliability becomes fundamental. Many Task Computing (MTC) is the latest emergent trend and identifies a heterogeneous set of applications and requirements for applications, which fill the gap between HPC and HTC.

We have briefly reviewed common models related to task programming. Embarrassingly parallel applications are composed of a collection of tasks which do not relate to each other, can be executed in any order, and do not require co-allocation. Parameter sweep applications are a special instance of embarrassingly parallel model. They characterized by a collection of independent tasks which are automatically generated from a template task by varying the combination of parameter values. In this case, the task executed is the same in terms of computation logic, but operates on different data. Therefore parameter sweep application can also be considered an expression of the SPMD (Single Program Multiple Data) model. MPI applications are characterized by a collection of tasks that need to be executed all together and which exchange data by message passing. Even though the program executed by an MPI application might be the same, it is quite common to provide an implementation logic that differentiates the behavior of each task according to its rank. Workflow applications are characterized by a collection of tasks whose dependencies can be expressed in terms of a directed acyclic graph. Dependencies are mostly represented by files, which are produced as output of a specific task, and are required for the computation of the dependent tasks. The nature of the tasks and kind of computation performed by each task is in general different.
We have finally introduced the Task Model and the services implemented in Aneka supporting task based programming, as a practical example of a framework enabling the development and the execution of distributed applications based on tasks. The Task Model comprises a set of services (directory, scheduling, execution, and storage) whose coordination constitutes the runtime support for the execution of embarrassingly parallel applications. The fundamental features of the task model in terms of task definition, submission, execution, and file dependencies management has been demonstrated with a practical example. On top of this infrastructure, client side components and integration with other technologies allow providers to support parameter sweep and workflow applications. Parameter sweep applications are realized through the Parameter Sweep Model (PSM), which is characterized by a collection of client side components providing different, and more suitable, interface for this kind of applications. Workflow applications are not natively supported by Aneka, but integration with other technologies allows leveraging Aneka for workflow execution. For example, a plug-in using the Aneka task submission web service allows the Workflow Engine to use Aneka as a backend for workflow execution. The Aneka distribution engine implemented in Offspring provides another example on how it is possible to quickly prototype another programming model (in this case a workflow based model) by leveraging the base APIs of Task Model.

7.5 Review Questions

1. What is a task? How does task computing relate to distributed computing?
2. List and explain the computing categories that relate to task computing.
3. What are the main functionalities of a framework supporting task computing?
4. List some of the most popular frameworks for task computing.
5. What does the term Bag of Tasks mean?
6. Give an example of a parameter sweep application.
7. What is MPI? What are its main characteristics?
8. What is a workflow? What are the additional properties that this application model has with respect to an embarrassingly parallel application?
10. How does Aneka support task computing?
11. What are the main components of the Task Programming Model?
12. Discuss the differences between ITask and AnekaTask.
13. Discuss the differences between static and dynamic task submission.
14. Discuss the facilities and the general architecture provided by Aneka for movement of data for task based applications.

15. How it is possible to run a legacy application by using the Task Programming Model?

16. Does Aneka provide any feature for leveraging the Task Programming Model from other technologies and platforms?

17. By using the Task Programming Model, design and implement a simple application that performs the discrete computation of the integral according to the method proposed by Riemann of a given function over a specified interval.

18. What are the features provided by Aneka for the execution of Parameter Sweep applications?

19. Does Aneka provide native support for the execution of workflows?

20. By taking as a reference the Montage Workflow described in Figure 7.6 design a sketch of the control flow of an Offspring strategy that can be used to execute a workflow on Aneka.