Hierarchical resource management in grid computing

Korkhov, V.V.

Publication date
2009
Document Version
Final published version

Chapter 6. Data-driven Workflow Management on the Grid

6.1 Introduction

Grid brings the power of many computers to scientists. However, the development of Grid-enabled applications requires knowledge of Grid infrastructure and low-level API to Grid services. Workflow management systems provide a high-level environment for rapid prototyping of experimental computing systems. Coupling Grid and workflow paradigms is important for the scientific community: it makes the power of the Grid easily available to the end user. The paradigm of data driven workflow execution is one of the ways to enable distributed workflow on the Grid.

In this chapter we discuss the highest layer of the complex distributed application hierarchy proposed in Chapter 1: a Grid workflow. Here we concentrate on the workflows controlled by a dataflow: discuss a model of a data-driven workflow and evaluate different strategies of resource management for this type of workflows.

To be executed on the Grid a distributed workflow needs a framework that enables the workflow enactment and execution control: a workflow management system (WMS). In this chapter we present the VLAM-G WMS that was developed to compose, execute and study data-driven workflows in a distributed environment. The core component of VLAM-G is the Run-Time System (RTS). The RTS is a dataflow driven workflow engine which utilizes Grid resources, hiding the complexity of the Grid from a scientist. Special attention is paid to the concept of dataflow and direct data streaming between distributed workflow components. We present the architecture and components of the RTS, describe the features of VLAM-G workflow execution, and evaluate the system by performance measurements.

Being the top layer of the complex application hierarchy, a workflow encompasses the applications of lower layers, introduced in the previous chapters, as its components. The management of these different layers is performed within a single framework, and the execution of each component is coordinated with the others. We illustrate this with a sample workflow for the driving application - the Virtual Reactor.

This chapter is based on: V. Korkhov, D. Vasunin, A. Wibisono, A. Belloum, M. Inda, M. Roos, T. Breit, L.O. Hertzberger. "VLAM-G: Interactive Dataflow Driven Engine for Grid-enabled Resources", Journal of Scientific Programming vol.15(3), pp. 173-188 (2007), ISSN 1058-9244

6.2 Data-driven workflows in a virtual laboratory

Grid environment allows coordinated resource sharing and problem solving among groups of trusted users within Virtual Organizations. Such environments enable global distributed collaborations involving large numbers of people and large scale resources, and make data and computing intensive scientific experiments feasible. Complex scientific experiments often require access to distributed resources such as computational resources, data repositories, third-party applications, scientific instruments etc. The utilization of these resources requires multi-domain expertise which is beyond the common knowledge of a single scientist.

One of the important research topics in e-Science [48] is to develop effective Grid enabled Problem Solving Environments and Virtual Laboratories for different scientific domains. Organizing software utilities (e.g. simulators, visualization and data analysis tools) as a meta experimental environment, a Virtual Laboratory allows a scientist to plan and conduct experiments at high level of abstraction [41] and enable a group of researchers located around the world to work together on a common set of projects. To execute and control experiments within a a Virtual Laboratory that typically consist of a number of interdependent simulations and other activities Scientific Workflow Management Systems (SWMS) are often employed[20, 85]. A SWMS explicitly models the dependencies between experiment processes, and orchestrates the runtime behavior of involved resources according to a flow description.

A wide range of e-Science applications from different scientific domains, namely high energy physics, bio-informatics, bio-diversity, bio-medicine, and tele-science [2] shows that data access and processing play an important role. A typical scenario involves a number of steps: (a) access data on a remote storage system; (b) move the data to one of the available computing nodes; (c) process the data, which results in a new data set (intermediate data) that is further processed on another node and thus needs to be moved again. These steps are repeated until the final data sets are generated and stored in a predefined location.

Frequently, when following this scenario, scientists would prefer to delegate all non-scientific activities such as finding the appropriate available resources and manipulating input, output, and intermediate data sets to a system they trust. Hiding such details of the underlying grid execution from the end user allows them to concentrate on the essential tasks they need to perform in order to achieve their scientific goals, e.g. defining original data sets, defining processes that need to be performed and the order in which these processes must be executed. On the other hand, the in-depth view at underlying processing on the low-levels can be provided as well, and an interested user can check what particular resources are used, examine intermediate results of execution and monitor actual computational processes. The ability to control one or a few parameters of the processes interactively, without having to stop or restart experiments, is also desirable.

In the scenario above, as well as in many other scenarios describing scientific experiments such as weather predictions [42], bio-medical visualization [69], knowledge discovery in databases [63, 90], and other scientific domains [90], data is one of the main drivers of progress in these experiments. In the paper [19] the authors describe

three different use cases from chemistry, cosmology and astronomy, where different patterns of pipelined processes have to be executed concurrently on distributed computing resources. For performance reasons data is streamed between the components composing the different levels of the pipeline. As described in [36], processing data in streams has certain advantages because it can be done on the fly, reducing the need to store intermediate data which is not needed.

The analysis shows that the computational processes in many e-Science experiments correspond to the actual data flow in an experiment; typically from an instrument, through analysis software, to data storage facilities. Consequently, the computation processing can be represented by a data driven workflow.

To efficiently execute large scale workflows in distributed environment, scalability support on different levels is needed. The workflow engine should be scalable enough to enable the execution of workflows consisting of a large number of components, possibly sub-workflows running cross geographically distributed resources. Support for a scalable job management is needed, especially for job farming and parameter sweep jobs. To ensure the efficient execution and utilization of resources a proper resource management is needed on multiple layers: starting from parallel tasks and up to the whole functional decomposition of the application on the workflow level. Finally, the introduction of semantic descriptions enables a semi-automated discovery and matching of workflow components.

Current trends in the research and development of workflow management systems has already been outlined in Section 2.4. Nowadays a number of scientific workflow management systems are developed in various research projects. In particular, some of the most mature frameworks are Triana [108], Taverna [92] and Kepler/Ptolemy [6]. All these workflow management systems have their own features and specifics, e.g. Taverna is totally dedicated to Web service composition highly demanded by contemporary bioinformatics; Triana provides interfaces to invoke web services and use Grid resources (GAP, GAT [5]) and has a rich library of processing modules; Kepler inherits a powerful and matured framework from Ptolemy, it provides a rich library of actors, Nimrod support and several types of workflow execution. An extensive overview of current workflow systems is given in [30].

The workflow management system discussed in this thesis, VLAM-G (Virtual Laboratory AMsterdam on the Grid) developed in the context of the Virtual Laboratory for e-Science project [140], has a set of distinguished features. VLAM-G provides a basic set of capabilities for building workflows by connecting components to each other based on data dependencies. As a core concept VLAM-G uses dataflow between simultaneously executing distributed components as an execution driving activity, while many other systems employ control flow and/or only perform sequential execution of workflow steps according to intermediate data readiness. Intermediate data handling is also performed differently. Some systems like Triana and Taverna use a centralized approach whereby all the data is controlled and collected at the central point where the engine resides. In turn, in VLAM-G data are transferred directly between participating components. Another distinguishing feature of VLAM-G is its ability to control an executing workflow at runtime in several ways. Firstly, the workflow components can be parameterized; the parameters can be controlled during execution.

Secondly, as VLAM-G operates with remote jobs co-allocated at runtime, it enables direct access to the graphical output of a component (i.e. component GUI, if one exists) by providing a shared virtual display. To the best of our knowledge no other system provides similar capabilities.

6.3 Resource management for data-driven workflows

6.3.1 Workflow modeling

Most Grid resource management systems being developed aim at a specific class of applications. In particular they can utilize different program and performance models as well as scheduling policies. The models for current high-performance schedulers generally represent a distributed application in a dataflow style or by a set of application characteristics. A performance model provides an abstraction of the behavior of an application on an underlying system and calculates predictions of application performance within a given timeframe. Current high-performance schedulers employ a wide spectrum of approaches, which differ in terms of who supplies the performance model: the system, the programmer or some combination of both. The performance model is commonly parameterized by both static and dynamic information. The scheduling policy ensures the achievement of the application performance goal. A usual goal is to minimize the execution time although different parameters can be optimized (e.g. execution cost). In many current efforts the scheduling policy is to choose according to a performance model the best set of resources among the candidate resource choices.

Consider a simple model of a distributed workflow (based on the meta-application model described in [115]). A workflow consists of a set of components that communicate with each other during the application run-time. The components are separate tasks that can be computational or data-processing units, interfaces to specific devices, data storage interfaces etc. Some of the components may be schedulable (like computation or data processing) and some are fixed (as data storage or remote device). In general, fixed components may affect the placement of the other components (i.e. a data processing component requires high speed link with a fixed database component).

Workflows can be represented as a data-flow graph. In the simple approach they can be divided in two categories: concurrent and pipeline. Concurrent application consists of a set of components running on different sites concurrently and exchanging data at the same time (Fig. 6.1-a). Pipeline application components start processing one after another and not concurrently, in a chain-like fashion (Fig. 6.1-b). A mixture of these types may also exist.

The scheduling of a workflow requires a cost model to evaluate the efficiency of a candidate schedule. The cost functions depend on the information about the application.

Consider application component $C_i, (i \leq N)$ where N is the number of components

a) concurrent b) pipeline

Figure 6.1: Basic types of data-driven workflows

(for simplicity here we consider all components as schedulable). For each component we define:

- $comp(C_i)$ - the computation load of the component C_i, may be counted in the number of instructions to be executed;
- $comm(C_i, C_j)$ - the communication load between C_i and C_j, may be counted in the number of bytes transferred between the components;

Let us now consider $S_k, (k \leq M)$, M - number of sites. Thus we can define:

- $compT(C_i, S_k)$ - the computation time for the component C_i running on the site S_k when the site provides all its resources to the application. In case other components are scheduled in S_k then the function may be degraded.
- $commT(C_i, C_j, S_k, S_l)$ - the communication time taken for data transfers between C_i scheduled on S_k and C_j scheduled on S_l . The function may depend on the number of components sharing the link.
- $commTT(C_i, S_k)$ - the total communication cost for component C_i placed on site S_k , is a function of $commT$ functions for the component's links. In the simplest case is the sum of those costs.

For a candidate schedule we get a set of pairs $(i, k) \in P_q$, that assigns components C_i to resources S_k , where P_q describes a particular candidate schedule; $q \in Q$, where Q is a set of all estimated schedules.

Thus we can define cost functions for a workflow as a whole in terms of the component cost functions. For concurrent applications we get:

$$T_c = \underset{(i,k)\in P_q}{max} [compT(C_i, S_k) + commTT(C_i, S_k)]$$

For pipeline workflows we get:

$$T_p = \underset{(i,k)\in P_q}{sum} [compT(C_i, S_k) + commTT(C_i, S_k)]$$

The scheduling problem is to determine such an assignment of components to resources that minimizes the cost function T for the workflow.

6.3.2 Heuristic algorithms for workflow scheduling

To create an optimal schedule of a workflow means to assign all the tasks composing the application to appropriate resources to minimize the cost of execution, usually counted as overall execution time. The model for estimation of the cost (performance metric) was presented in Section 6.3.1. Here we address the algorithms used to achieve sub-optimal schedules based on the abovementioned performance metric.

Each component in the modeled workflow is provided with the execution requirements for processing: computational load (CPU cycles needed), communication load (data transfers size), memory and storage requirements. The information on the characteristics of the available resources (CPU speed, available memory and storage, network links speed) can be retrieved from the Grid information services, thus the target is to perform the matchmaking of the workflow components and resources to ensure the optimal execution. We propose and evaluate the following algorithms to create the schedule for the workflow execution: two types of basic greedy algorithm, computation-network prioritized algorithm, and simulated annealing.

Basic greedy algorithm (BG):

This is the straightforward version of the greedy heuristic we propose and use as the starting point of the experiments. We assume that a resource used to execute a workflow component can be shared between several components or even utilized by other background jobs, thus the component is not in the full possession of computing resources and only a share of computing power is available. This heuristic consists of the following steps:

```
Reqs = get_requirements(WFcomps) // Find out the minimal requirements
               // for the workflow components (set of minimal memory
               // and storage requirements)
Res = discover_resources(Reqs, GIS) // From the Grid indexing service
               // discover a set of available resources satisfying the
               // minimal requirements
sort_descend(Res, CPU) // Sort the created resource pool according to
               // available CPU power
sort_descend(WFcomps, Reqs.CPU) // Sort all the workflow components
               // according to CPU requirement
compsPerRes = 1 // Maximal number of components allowed to be mapped
                // on the same resource
for comp in WFcomps do // Staring from the component with highest CPU
                       // requirement iterate all the components
  comp.mapped = false
  while(!comp.mapped) do
    for res in Res do // Iterate all the resources available in the
                      // pool beginning from the most powerful one
      if(res.fits(comp.reqs) && (res.alreadyMapped < compsPerRes))
      then // resource fits (memory/storage requirements satisfied
           // the number of already mapped components does not
```

```
        res.map(comp)                          // exceed the limit)
        res.alreadyMapped++       // map this component
        comp.mapped = true        // to this resource and
        break                     // stop iterations
      end if
    end for
    if(!comp.mapped) then
      compPerRes++       // If all the resources have been
                  // iterated and none could fit then increase
                  // permissible number of components per resource
      if(compPerRes > WFcomps.size()) exit(NO_MAPPING_FOUND)
    end if
  end while
end for
```

Modified basic greedy algorithm (BGM):

The basic greedy algorithm is modified so that the same resource could be used for executing several components simultaneously unless it decreases the overall performance of the workflow compared to the case when all the components are placed to separate nodes. Matchmaking for each component is changed in the following way:

```
for comp in WFcomps do
  for res in Res do                  // Perform the estimation of the
    if(res.fits(comp.reqs)) then     // run-time (Section 6.3.1) for all
      res.map(comp)                  // the available resources, select
      T=estimate_runtime()           // the optimal one (the one with
      res.unmap(comp)                // minimal run-time) and map the
      if(T < bestfitT) then          // component to this resource
        bestfitT = T; bestfitRes = res
      end if
    end if
  end for
  bestfitRes.map(comp)
end for
```

The main difference with basic greedy algorithm is that here the components can be scheduled to the same resource before all the resources are occupied with at least one component. This can be efficient in the case of a single powerful resource among a set of much slower resources.

Computation-network prioritized algorithm (CNP):

This algorithm introduces a way to describe the level of a relative workflow intensity of data transfers and computations, which is a reflection of the Advanced Workload Balancing (AWLB) algorithm introduced in Chapter 3 and was used for resource ranking

in User-Level Scheduling environment (Section 5.2.4). Again, we use the parameter f_c which defines priority either of networking communications or computational operations for the experiment.

The f_c parameter affects the calculation of the resource rank which is used in resource sorting and selection procedures. We define 'the base resource' as the resource for the component with the highest requirements - the most powerful of the available resources. The rank of the resources is calculated using the bandwidth to the base resource and available computing power of the resource. The heuristic formula for resource rank is similar to the one used in Section 5.2.4:

$$r_i \sim p_i(1 + f_c/\mu_i) \tag{6.1}$$

where r_i is the rank of resource i, p_i - processing power of resource i, and μ_i is the resource capacity parameter. Here $\mu_i = p_i/n_i$, where n_i is the bandwidth between the base resource and the resource i.

The f_c coefficient here reflects an aggregate average communications/computations characteristic of the whole workflow. By varying the value of f_c different resource priorities can be achieved and different resulting schedules can be obtained. Compared to the adaptive workload balancing algorithm presented in Chapter 3, the optimal value of f_c can be obtained using a similar procedure as described in Section 3.3.

Simulated annealing algorithm (SA):

Simulated annealing is a technique, which was developed to help solve large combinatorial optimization problems [1]. It is based on probabilistic methods that avoid being stuck at local (non-global) minima. It has proven to be a simple but powerful method for large-scale combinatorial optimization.

In the simulated annealing algorithm an objective function to be minimized is defined. Here it is the overall execution time of an experiment. The execution time of each workflow component is equivalent to the "energy" of the component. Then, "temperature" is the average of these times. Starting from some initial schedule and initial temperature, the algorithm randomly selects a component to be re-mapped, randomly selects a suitable resource and re-maps the component. The total "energy" (execution time) of the experiment is estimated. Any downhill step is accepted and the process repeats. An uphill step may be accepted. Thus, the algorithm can escape from local minima. This uphill decision is made by the Metropolis criterion. The Metropolis criterion attempts to permit small uphill moves while rejecting large uphill moves. To allow this a uniformly distributed random number p in the range [0...1] is generated and compared with the Metropolis number:

$$m = \frac{e^{delta}}{temperature}$$

where 'delta' is the difference in energy between current schedule and candidate schedule. Since delta/temperature is negative (candidate schedule energy is larger then

current one), e is raised to a negative power so that m will also be in the range [0...1]. The decision to accept an uphill move is made if randomly generated number p is less than Metropolis number m. The process of finding an optimum is divided to a number of series of iterations (during each iteration series a number of re-mapping steps are performed) with decreasing temperature on each step. Decreasing values for temperature as well as large negative values of delta make acceptance of an uphill move less likely. As the optimization process proceeds, the algorithm closes in on the global minimum. Since the algorithm makes very few assumptions regarding the objective function, it is quite robust. The degree of robustness can be tuned by the user by adjusting the following parameters:

- factor - annealing temperature reduction factor
- ntemps - number of temperature steps to try
- nlimit - number of trials at each temperature
- glimit - number of successful trials (or swaps)

Pseudo code of a basic process:

```
initialize temperature

for i := 1...ntemps do
  temperature := factor * temperature
  for j := 1...nlimit do
    calculate current_cost
    trial_cost := randomResourceSwap()
      // swap resources randomly
    delta := current_cost − trial_cost
      // cost value change

    if delta > 0 then
      make the swap permanent
      increment good_swaps
    else
      p := random number in range [0...1]
      m := exp( delta / temperature )
      if p < m then
        // Metropolis criterion
        make the swap permanent
        increment good_swaps
      end if
    end if
    exit when good_swaps > glimit
  end for
end for
```

6.3.3 Simulation results and discussion

For the experiments we allocated a resource pool of 20 available machines of various computing power, memory and storage capabilities. The machines were distributed across 5 domains with different bandwidths within the domains and between the domains. We combined the domains with powerful computational resources linked by a low bandwith links with faster connected, but slower in performance resource domains. All the mentioned algorithms were tested in this environment on several experiment topologies consisting of various number of components: from 3 to 10. Figure 6.2 presents the results for a set of experiment topologies.

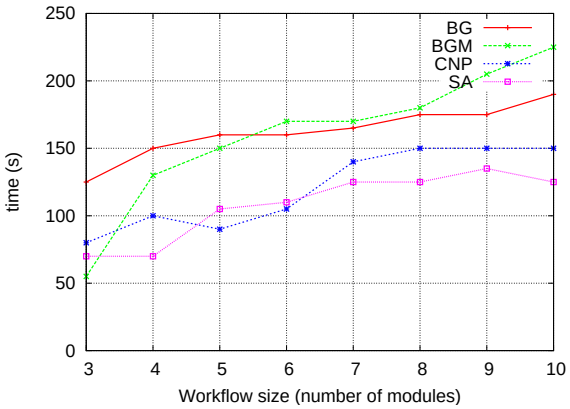

Figure 6.2: Simulation results of scheduling performance for different scheduling algorithms. The dependency of the workflow runtime on the amount of components executed.

We can see that the basic greedy algorithm usually gives one of the two worst results. The modified basic greedy algorithm has different behaviors: in topologies with small number of components it is very efficient and gives good results, though the more the size of experiment grows the less efficient it becomes. For the topologies with more than 6 components BGM gives the worst results among all the algorithms. Computation-network prioritized algorithm is examined with the value of f_c coefficient equal to 0.3, which proved to be the best for the test topologies during a separate experiment. In all cases it gives better results than basic greedy algorithms (both standard and modified) only except the case of 3 components when BGM is the most effective. The best results (except 3 components case) have been shown by simulated annealing algorithm, though it was the most time consuming one.

The simulation results have shown that heuristic mapping algorithms might be efficient in some system configurations, especially in small homogeneous environment, but for generic case and complex system topologies the algorithms of random search like simulated annealing are more promising. The drawback of such algorithms is the requirement for increased processing time caused by much more complex and

extensive computations taken. Typically, the time consumed by SA for scheduling the experimental topologies was around an order of magnitude higher that the time required by the straightforward heuristic algorithms, though the absolute value can still be neglected in the scale of workflow execution time: typical time for heuristic algorithms was limited by 1 second while SA took at most 10 seconds during the experiments.

6.4 VLAM-G: interactive data driven workflow management system for the Grid

6.4.1 The vision

The aim of the VLAM-G system is to provide and support coordinated execution of distributed Grid-enabled components combined in a workflow. This system combines the ability to take advantage of the underlying Grid infrastructure and a flexible high level rapid prototyping environment. On the high level, a distributed application is composed as a data driven workflow where each component represents a process or service on the Grid. Processes are activated only when the data is available on their input ports. The significant difference from other similar systems is the support for simultaneous execution of co-allocated processes on the Grid which enables direct data streaming between the distributed components: traditional batch processing of grid jobs and workflow execution based on input/output files exchange between the components is not suitable for many use case scenarios. This feature is highly required for semi-realtime distributed applications e.g. in the bio-medical domain or in online video processing and analysis [101].

Grid technology is maturing very quickly, the new paradigm based on SOA architecture is replacing the original resource oriented approach. However, the transition to the new paradigm will not take place overnight; moreover, the old approaches remain more efficient in a number of cases. One of the targets of VLAM-G is, thus, to enable support for co-existence of different types of grid execution models within a single workflow. This goal is achieved by abstracting a particular Grid execution model to an intermediate common representation. In VLAM-G a workflow is not composed of particular specific Grid jobs or services but of components with a special interface. These components are called modules, and they are the core entities of the VLAM-G data-driven workflow. Thus a module can represent a specially developed application which uses the VLAM-G native module API (`VLport`), a web service, or a legacy application.

The runtime control of the execution of a distributed workflow provides the ability to monitor the execution and influences the behavior of workflow components. VLAM-G supports several ways of runtime control: direct interaction with the user interface of a module (remote X GUI access) and module parameter control (reading flags and values set by a module and updating these values from outside the module). Monitoring delivers all the log data from remote modules to the VLAM-G user interface thus all the issues in module execution can be tracked centrally.

Intensive distributed data processing might take a long time. To facilitate the handling of the executing workflow, the system is capable of closing the user interface, detaching from the workflow engine and re-attaching later on during runtime.

The core features of the system we present are:

(1) Dataflow (and not control flow) is used as a driving force;

(2) Workflow components (VLAM-G modules) are versatile: a module may be either a specially developed software component (written in C++/Java/Python using the VLAM-G API), or an interface to legacy applications or web-services;

(3) Distributed execution: support for Grid job submissions together with web services and local tasks within a single workflow;

(4) Support for legacy applications wrapped as modules (flexible XML configuration);

(5) Support for remote graphical output: remote X display for Grid jobs is provided;

(6) Interactivity support: online control via parameters, and via remote graphical output;

(7) Decentralized handling of intermediate data;

(8) Decoupling of GUI and engine;

6.4.2 The architecture

In this section we present the architecture of the VLAM-G workflow engine which enables the execution and the runtime control of distributed data-driven workflows on the Grid. A workflow is composed of a set of components called modules which represent an executing entity (remote application or a web service). As computing resources VLAM-G can use: (1) grid enabled local or remote sites (so far VLAM-G works only with the Globus middleware) that enable inbound and outbound communications; this allows data streams to be established between workflow components located on remote grid resources; (2) web and grid services. Below we will concentrate on the architecture of the workflow engine starting from modules as workflow components and finishing with the description of the components the engine is built of.

The VLAM-G environment is constructed from two core components: the graphical user interface (VL-GUI) and the Run-Time System (RTS) – the engine which prepares and executes the workflow, handles the intermediate data and allows the monitoring and the online control. The VL-GUI is used to create a complex scientific experiment interactively by composing a workflow, setting the parameters for each workflow component before and at runtime, and transferring the workflow description to the engine using standard SOAP protocol.

In the following sub-sections, we describe the architecture of the RTS which consists of the RTS Manager (RTSM), the RTSM factory, and the Resource Manager.

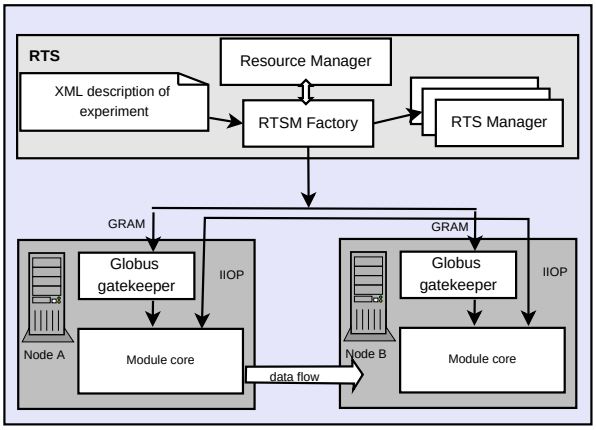

Figure 6.3: Run-Time System Architecture.

The workflow modules

Modules are the core composition elements of the VLAM-G framework. A module is an entity which represents a task to be executed on the Grid. It can be a specially developed application, a web service, or a legacy application. A workflow is composed of a set of modules. The modules have input and output ports which are used to compose the workflow by connecting the output of a module to the input of other modules.

VLAM-G workflows often target coordinated and simultaneous distributed processing of streaming data. Assuming this, a model of a module can be represented as the sequential retrieval of data from input ports, data processing and distribution of the result data to output ports. The execution is performed in a continuous fashion, so that these steps are repeated until interrupted or until the incoming data stream is closed.

A module represents a task being executed on the Grid and from the design point of view it consists of a processing part and a service part. The processing part represents the application itself: the code developed using the VLAM-G module API, a call of a web/grid service, or a launch of a legacy application. The service part provides the interface and the basic facilities for runtime control and the management of data transfers between modules. All available modules are stored in a repository, the creation and deployment of a new module is simple and straightforward and will be explained in the course of the chapter.

The Run-Time System architecture

Figure 6.3 shows a high level overview of the VLAM-G workflow engine: the RTS. The main components of the system are the RTSM Factory, the RTSM and the Resource Manager (RM). The scenario of a workflow enactment using the RTS is the following:

- The RTSM Factory creates an instance of the RTSM, the VLAM-G workflow engine. The RTSM instance is responsible for the given workflow execution. Multiple simultaneous executing workflows are supported. The engine takes as input a directed acyclic graph representing the dataflow of the workflow where the nodes correspond to the modules (workflow components) and the edges reflect the data streams.

- The computing resources where the modules are scheduled can be specified explicitly or retrieved from the RM. If the execution host of a module is not specified then the RM searches for optimal resources for executing the workflow by utilizing the information on module resource requirements. The RM handles the discovery, location, and selection of the necessary resources according to the VLAM-G module requirements. It maps the application tasks to the appropriate resources to optimize the workflow performance, utilizing a number of algorithms and scheduling techniques [65].

- After the resources have been selected, the workflow becomes fully concrete, and the RTSM schedules the workflow components using the Globus job submission mechanism (with non-web service modules), connects the module ports according to the workflow description and sets the module parameters to their default values.

- The RTSM starts the workflow and monitors the execution of each of the VLAM-G modules in this workflow.

VLAM-G resource management

To create an optimal schedule for a data-driven workflow we need to assign all the components composing the workflow to appropriate resources with the goal of minimizing the cost of the workflow execution. For this, each module in the VLAM-G framework is provided with a module description file that contains information about module resource requirements, including a default execution location which may be left blank.

Before creating an RTSM instance, the RTSM Factory contacts the RM. In turn, the RM processes the description of the workflow, with execution location missing for some modules, together with the requirements of the modules. The RM performs scheduling decisions based on the application information, the available resources information, and cost and application models (Figure 6.4). The application information includes the requirements that define the quality of the service requested by the modules. These requirements include the amount of memory needed, the approximate number of processing cycles (i.e. processor load), the storage and the communication load between modules. VLAM-G uses a RSL-like language to specify these requirements (RSL is a resource specification language used in the Globus toolkit to specify the job to be submitted to the Grid Resource Allocation Manager, [21]). The resource information is obtained from the Grid Monitoring and Discovery Service (MDS) [21], which also provides forecasts of the resource state from the Network Weather Service

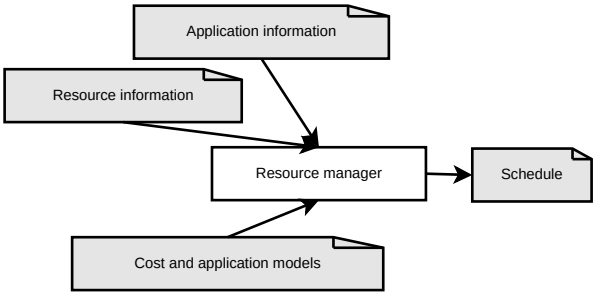

Figure 6.4: Resource Management in VLAM-G: Application and resource information along with cost and application models are processed by the resource manager to produce an execution schedule.

(NWS) [117]. This helps to estimate the resource load in the specified time frame in the future and to model the application performance. The cost and application models are used by the resource manager to evaluate a set of candidate schedules for the application. The cost model refers to the execution time of the workflow as the performance metric, and the application model specifies how the execution of the workflow is simulated. The application model is usually represented as a DAG which can be executed in a pipeline or concurrently, and the execution performance depends on the application and on the resource information.

The RM uses several types of heuristics and simulated annealing to achieve sub-optimal schedules based on a performance metric (generally the overall execution time is evaluated). For simulated annealing the RM tries a number of candidate schedules and simulates the execution of the workflow with given requirements on given resources. The results are estimated using a cost model, resource state information, and predictions. The algorithm converges to a sub-optimal schedule which is transferred to the RTSM Factory. Different types of meta-scheduling algorithms (heuristic algorithms and simulated annealing), the evaluation results, and analysis are discussed in detail in [65].

VLAM-G interactive module control

Each VLAM-G module may have parameters that can be monitored and changed during runtime by the module itself or by the user. A parameter in the VLAM-G context is a named entity with a value that can be changed either from within the module code or from outside, i.e. from the VL-GUI. A VLAM-G parameter can be compared to an environment variable of an operating system. This feature allows the users to interact directly with every module composing the application workflow, thus the execution of modules can be controlled on the fly without the need to stop and restart the whole workflow. A parameter is usually associated with a metric that needs to be controlled or is used to monitor the internal state of a module during execution.

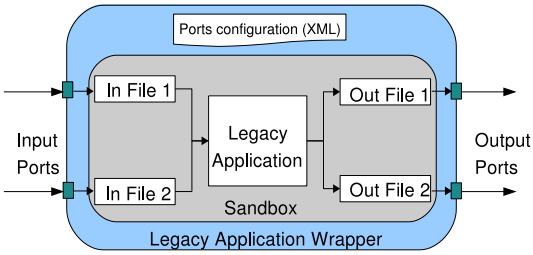

Figure 6.5: Legacy application wrapper: legacy applications operate input and output files that are mapped to VLAM-G ports by the wrapper. The wrapper is generic and can be configured using an XML configuration file.

Legacy application support

We define a legacy application (LA) as follows: LA is an application that has no source code available and cannot be modified. LAs are modeled as black box applications exchanging data files; both the input and output data needed for the execution of a LA is assumed to be a set of files. This model can be mapped directly to the VLAM-G module concept because input and output ports can be bound to the exchanged files. For the legacy applications, a generic wrapper has been developed (Figure 6.5).

Each LA has a configuration file where all input and output data files are described and bound to the input/output ports. The LA wrapper allows the RTSM to instantiate the LA in the same way as it instantiates a native VLAM-G module. The LA wrapper reads the data from input ports and stores them in the sandbox as files, then executes the LA code. The LA produces a set of output files stored again in the sandbox after processing the data. After the LA finishes the execution, the output files are sent to the associated output ports. This process is repeated until the stream of input files is over. Thus the processing is performed in a loop fashion: continuously repeating the retrieval of input files, the processing by the LA and the transfer of the output files, thus emulating the streams of input and output data.

The same technique is also applied to RPC style web services. Web services expose their operations in WSDL description. Each operation defined in this WSDL can be imported as a VLAM-G module by applying a similar approach as the one used by an LA. Thus a web service with several operations can be imported as a collection of VLAM-G modules. In turn, each of the parameters of every operation described in the web service WSDL is imported as an input port of the corresponding module, and its return values are described as output ports. VLAM-G provides a generic wrapper for this kind of web service operations The web service wrapper collects data from input ports and stores it in input files. A generic client for web services performs the web service calls, setting up operation parameters based on input files, and stores the result in an output file. Finally this output file is sent to the next module via the output port.

Other projects also address the problem of using legacy code in Grid environment. GEMLCA [31] enables the deployment of legacy code applications as Grid services

without the need of code re-engineering. Similarly to VLAM-G, the legacy code here is also provided as a black box with specified input and output parameters and environment requirements. The difference is that VLAM-G wraps an LA not as a web-service but as an entity supporting implicit data streaming.

6.4.3 VLport library: design and implementation

To provide implicit data exchange between the modules (remember that modules are not aware about existence of each other, they only read and write data through ports) a special connectivity layer was required. This layer maps abstract ports of modules to standard networking concepts (as sockets) and ensures that the modules are connected according to a workflow description and can communicate. Thus the `VLport` library was developed. The `VLport` library is responsible for maintaining the stream of data from an output port of a module to an input port of another module running on a different computing node. The library is a key component in moving intermediate data produced at a given step of the application workflow to the next step, regardless of the physical location of two processes.

The `VLport` provides the runtime environment for any VLAM-G module. It offers a basic API for the creation of I/O ports, changing the parameters of the VLAM-G modules, and other utility functions. From the implementation point of view the library provides a class, which must be used as base for developing any VLAM-G module. A module developer implements the abstract method `vlmain` (Figure 6.6 and listing 6.1), which contains all computational logic of the module. The RTSM instantiates, connects, and executes all modules composing the workflow. The input and output ports have a simple interface compatible with standard C++ streams. Legacy applications and web services are wrapped and represented to the workflow system as VLAM-G modules as well. Interfaces to Java and Python languages are provided so that native VLAM-G modules can be developed using these languages as well.

The library provides a number of utility functions that gives VLAM-G module developers easy access to all GASS data sources using GridFTP, FTP, HTTP and HTTPS protocol. The data streamed to/from ports can be serialized to the eXternal Data Representation (XDR). This makes it possible to exploit heterogeneous computational resources. However, module developers can also work with a raw stream, which is similar to the network sockets. The RTSM establishes all connections, a developer works with opened streams like any C++-compatible streams (i.e. iostream).

The RTS performs internal module control by using CORBA technology. CORBA has been selected as a stable and mature technology, not as resource-demanding as web services, to enable the management of centrally controlled distributed objects. For the RTS, each module is represented as a CORBA object [137]. Each instance of a module has IIOP CORBA reference and can be accessed remotely. In order to avoid firewall problems, the library chooses TCP ports from a range specified in GLOBUS_TCP_RANGE variable. Thus, a module can be instantiated even behind a firewall.

Listing 6.1: Example of a native VLAM-G module

```
#include <vlapp.h>
#include <fstream>

class MyVLApplication : public VL::VLApplication
{
public:
    MyVLApplication(int argc , char **argv)
        : VL::VLApplication(argc, argv), log("test_vlapp.log")
    {
        myOstream = createDefaultOPort("port_out");
    };
    virtual ~MyVLApplication()
    {
        delete myOstream;
    };
    int vlmain(int argc , char **argv)
    {
        std::ifstream fstr("File.orig.dat");
        time_t t1;
        time(&t1);
        *myOstream << fstr.rdbuf();
        std::cout  << "Time:" << time(NULL)-t1 << std::flush;
        return 0;
    };
private:
    VL::vostream  *myOstream;
    std::ofstream log;
};

int main(int argc , char **argv)
{
    globus_module_activate(GLOBUS_COMMON_MODULE);
    globus_module_activate(GLOBUS_IO_MODULE);
    try
    {
        MyVLApplication app(argc, argv);
        app.run();
    }
    catch(VL::Exception *e)
    {
        std::cerr << e->what();
        delete e;
    }
    globus_module_deactivate(GLOBUS_IO_MODULE);
    globus_module_deactivate(GLOBUS_COMMON_MODULE);
    return 0;
}
```

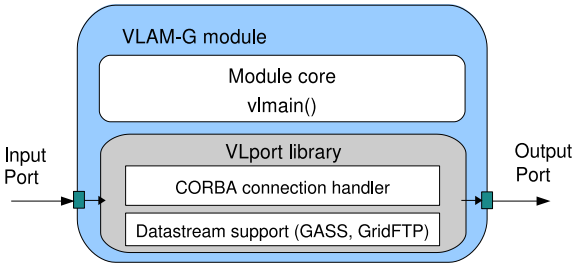

Figure 6.6: VLAM-G module structure: method vlmain() implemented by a module developer and supported by VLport library for transparent data transfers to other modules within a workflow.

The life cycle of a module can be described as follows:

1. *Initializing a module:* The RTSM instantiates a module on a grid-enabled (using the GRAM protocol) or local node. All necessary environment variables are set for proper module initialization.

2. *Creating input/output ports:* During the initialization stage the input and output ports are created. The number of ports is predefined for a module and can not be changed during the runtime.

3. *Registering a module:* The module contacts the RTSM and registers itself. The RTSM keeps sending keep-alive messages to the module during the runtime. If acknowledgment is not received during a predefined timeout the module is considered crashed.

4. *Connecting modules:* The RTSM connects modules with each other according to the dataflow; an output port can be connected with many input ports (one-to-many relationship);

5. *Scheduling a module:* The RTSM schedules the modules for execution (the method `vlmain` is executed);

6. *Exchanging data with other modules:* The module reads the data from input ports, processes it and writes results to output ports: loop fashion execution;

7. *Module termination:* When a module exits (i.e. input port has been closed by the previous module) it returns from `vlmain` method. All pending buffers are flushed, and ports are closed. The module unregisters itself from RTSM registry and exits.

The control of the parameters of a VLAM-G module is implemented as CORBA remote procedure calls. All the modules in the worklfow are controlled by the RTSM using IIOP protocol.

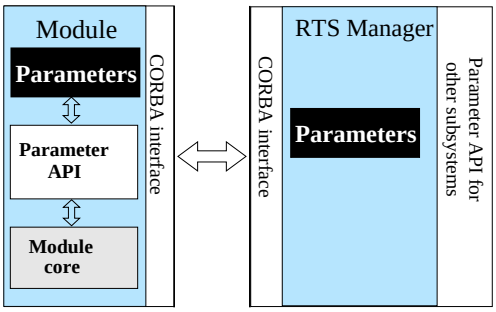

Figure 6.7: Module parameter interface: RTSM controls (sets and gets) parameters of a module via CORBA interface; module core controls parameters via specified API from VLport library.

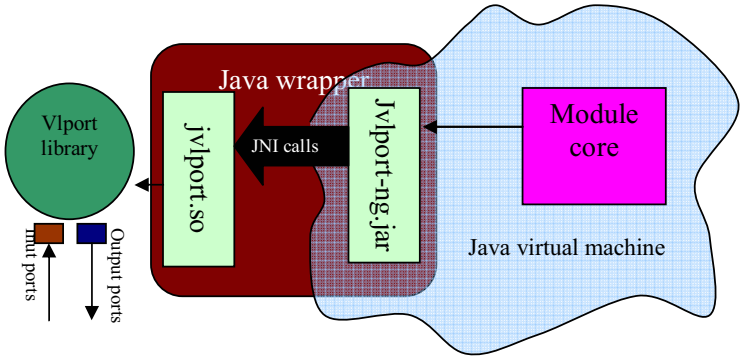

Figure 6.8: Java wrapper: module developers can implement modules in Java; Java calls are translated to C++ `VLport` library by the wrapper using Java Native Interface.

To utilize various programming languages such as Java and Python a set of wrappers has been developed. The wrappers translate the calls from the target language to the C++ `VLport` library. Figure 6.8 shows the Java wrapper library based on Java Native Interface (JNI). The VLAM-G module core and the wrapper are executed in the same Java virtual machine, the wrapper functions translate the calls to the `VLport` library. Thus a Java module has the same functionality and life-cycle as a native module. The Python wrapper employs similar techniques using the SWIG (simplified wrapper and interface generator) to generate a wrapper around the `VLport` library. Similar solutions for code wrapping have been employed in other projects, for example Jopera [94] provides a wrapper for Java snippets, small blocks of Java code usually needed to preform small computation. It also provides a wrapper for Unix applications (legacy applications) using the shell command line and the pipe-based inter-processes communication.

The VLAM-G port library is developed in C++ and is implemented as a dynamic library for UNIX systems. It uses the threaded versions of Globus libraries and

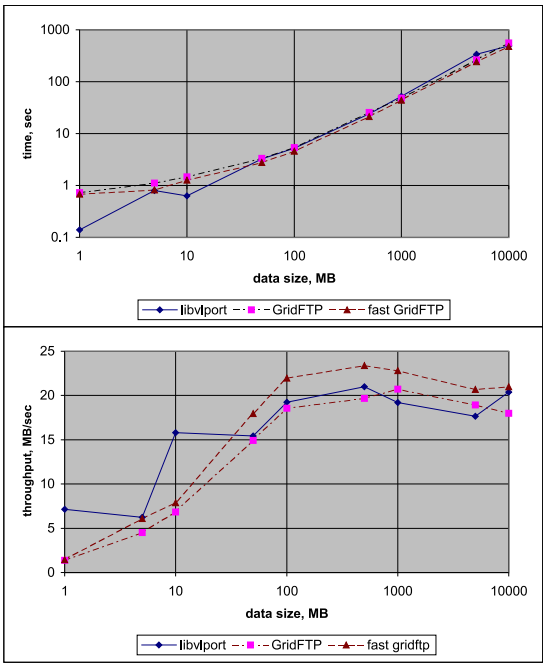

Figure 6.9: Average performance of the RTS library on WAN compared to standard Globus data transfer tool.

OmniORB - a free high performance ORB for C++ [137]. The RTSM is a part of the VLAM-G front-end and is developed in Java. It uses Commodity Grid Kit for Java [125] and Community OpenORB Project [126].

6.4.4 Performance evaluation

The RTS library is designed to provide maximal performance for distributed applications while hiding all low level details from the application developers. It must thus provide a throughput comparable with the standard protocols. In this section we compare the performance of the library with the performance of standard data transfer tools included to the Globus toolkit, and measure the introduced overhead.

We evaluated the dependency of the data transfer performance on the data block size using both standard Globus utilities and `VLport` library. The data transfers took place between the nodes of clusters connected through a high-speed WAN (wide area network). The clusters are part of the ASCI Supercomputer 2 (DAS 2), which is a cluster system distributed over different universities in the Netherlands [130]. To measure the overhead introduced by the VLAM-G library, the average throughput of the link was evaluated with the help of standard Globus tool 'globus-url-copy' using GridFTP protocol. Figure 6.9 shows the data transfer rate as a function of the data block size. Compared to the throughput obtained with the VLAM-G RTS library,

the globus-url-copy shows slightly better performance for the test with parallel stripes (fast GridFTP) and almost the same outcome in the case of a standard GridFTP protocol. The maximum data throughput for large data blocks is about 20 MB/sec for the library. Since the RTS Library is designed to support data streams, the performed tests show encouraging results as the speed of the data transfer achieved using the library is comparable to the one achieved using standard Globus tools.

6.5 Multi-layered application as a workflow

Chapter 3 outlined the challenges posed by the Grid to the multi-layered complex applications. We illustrated the generic method to deploy such kind of multi-component applications on a case study of the Virtual Reactor application. The proposed workflow for the Virtual Reactor experiment is shown in Figure 3.3.

Several approaches to the execution of the Virtual Reactor in high-performance computing environments have been examined. The initial version has been developed to be executed in a parallel computing environment via a Web portal interface [70]. Later, the Migrating Desktop was used to run the VR components on the Grid resources (section 3.2.2). But no solution was proposed to seamlessly integrate all the components into a single sequence of tasks that would not require the manual intervention and control on each stage of the VR workflow. Such a solution can be provided by a workflow management system as it is specifically designed for automatic sequencing of multiple tasks.

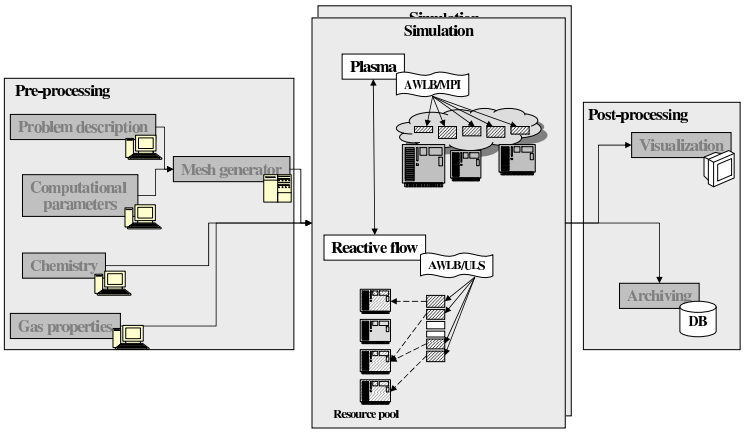

Figure 6.10: Multiple layers of processing within the Virtual Reactor workflow.

In the previous chapters we described the resource management and workload balancing for the components from different layers of a complex distributed application: starting from the task-level parallelism on heterogeneous systems in Chapter 3, its development on multi-cluster environment in Chapter 4 to job-level parallelism in Chapter 5. Each chapter covers a specific type of application execution in a dis-

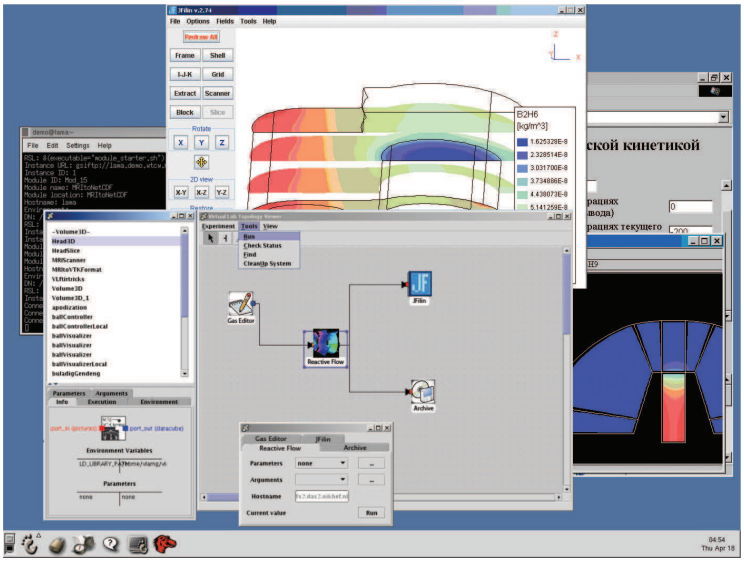

Figure 6.11: Virtual Reactor workflow in VLAM-G.

tributed environment, and typically each of these types can be used by a component of the complex distributed application, in particular the Virtual Reactor. A single workflow can include all these types of processing as its components. The workflow management system performs high-level resource management, leaving the particular management decisions for the components itself, thus the hierarchy is built: WMS assigns the components to computing systems, in particular homo- and heterogeneous clusters, and task management systems (like User-Level Scheduling). Figure 6.10 shows the Virtual Reactor workflow with the types of processing assigned to its components.

The prototype multilayer Virtual Reactor workflow has been implemented in the VLAM-G environment. The components are wrapped as VLAM-G modules. The workflow management system controls the data exchange between the modules (the standard way provided by VLAM-G is used instead of file transfers) and the sequence of modules execution. The results of processing can be archived or visualized in a virtual desktop provided by VLAM-G. The basic VR workflow is shown in Figure 6.11.

6.6 Conclusions

In this chapter we propose and compare several algorithms based on ad-hoc greedy model and on the random search technique that are adapted for data-driven workflow scheduling in the Grid environment. We study the behavior of these algorithms using simulation model of a Grid workflow. The results show that the heuristic greedy algorithms may be effective for mapping in specific network configurations especially

in small and homogeneous environment but for effective scheduling in heterogeneous environment more complex algorithms are needed. For the latter we propose using random search algorithms e.g. simulated annealing algorithm. These algorithms allow to avoid local minima and to get result closer to global optimum, but their drawback is the significant time needed to find sub-optimal schedule. Our experiments show that the random search approach is promising for scheduling in Grid environments.

As the implementation of a workflow management system we present the VLAM-G – a data-driven WMS, and the Run-Time System, its engine. This engine allows the execution of data-driven workflows in a transparent way on the available Grid resources. A VLAM-G workflow is composed of entities called modules that interface native VLAM-G applications, web services and legacy applications even if the source code is not available for modification. Interactive control of the execution is provided: the end-users can interact with any workflow component at runtime via a simple parameter interface or by accessing a virtual display with remote graphical output. The API for native VLAM-G modules is provided by `VLport` runtime libraries which have been designed to support different languages: C / C++, Java and Python. A generic wrapper provides the means to port an existing application or a web service as a standard VLAM-G workflow component.

The VLAM-G workflow management system can be used for distributed applications requiring direct data streaming between the remote components, e.g. semi-realtime applications, remote devices access and control etc. The performance evaluation showed that the overhead brought by intermediate `VLport` library is negligible.

As the top layer of the complex application hierarchy, a workflow embraces the components representing the application of lower layers of the hierarchy, introduced in the previous chapters. A single framework performs the management of these different layers, and the execution of each component is coordinated with the execution of the other components. The organization of such a composite workflow is illustrated with a sample workflow developed for the driving application - the Virtual Reactor.