Deliverable D2.3

Analysis on Elements of Cooperation between Programming Model Frameworks and Resource Management Systems

VERSION Version 1.0

DATE July 29, 2011

EDITORIAL MANAGER Christian Perez (Christian.Perez@inria.fr)

AUTHORS STAFF

Yann RADENAC	IRISA/INRIA, Rennes (MYRIADS team)
Alexandre DENIS	LaBRI/INRIA, Bordeaux (RUNTIME team)
Cristian KLEIN	LIP/INRIA, Lyon (GRAAL/Avalon team)
Christian PÉREZ	LIP/INRIA, Lyon (GRAAL/Avalon team)
Frédéric CAMILLO	INPT/IRIT, Toulouse
Ronan GUIVARCH	INPT/IRIT, Toulouse
Aurélie HURAULT	INPT/IRIT, Toulouse
André RIBES	EDF R&D, Paris

Analysis on Elements of Cooperation between Programming Model Frameworks and Resource Management Systems

Yann Radenac, Alexandre Denis, Cristian Klein, Christian Perez, Frédéric Camillo, Ronan Guivarch, Aurélie Hurault and André Ribes

Abstract

This reports first presents a taxonomy for classifying applications with respect to their interactions with RMS. Hence, applications can be static (being rigid or moldable) or dynamic (malleable or evolving).

Next, the cooperation between the two main programming model framework of COOP – GridTLSE and SALOME – and the two main resource management systems of COOP – Diet and XtreemOS is analyzed. Also, using SALOME on top of PadicoTM is studied.

Contents

1 Introduction

The COOP project aims to reconcile Programming Model Frameworks (PMF) and Resource Management Systems (RMS) with respect to a number of tasks that they both try to handle independently. Deliverable D2.1 has presented an analysis of existing PMF and in particular those developed by the partners of this project: GridTLSE (IRIT), SALOME (EDF-CEA) and HLCM (INRIA). Their main characteristics have been presented as well as the expected improvements in the framework of the COOP project. Deliverable D2.2 has presented an analysis of existing RMS and in particular those developed by the partners of this project: Diet (INRIA), XtreemOS (INRIA) and PadicoTM (INRIA).

This report analyzes cooperations mechanisms between the PMFs and RMSs that are part of the COOP project. It begins by presenting a taxonomy for classifying applications with respect to their interactions with an RMS in Section 2. Then, it presents and analyzes some particular use-cases that are going to be studied within the COOP project. The use-cases have been sorted with respect to the two large programming environments studied in COOP: GridTLSE and Diet.

Section 3 focuses on GridTLSE and its interactions with Diet. Two motivating examples are presented – threshold pivoting and multiple right-hand sides – as well their impact on Diet.

Section 4 describes various SALOME scenarios. First, it presents two applications based on SALOME – HLW and Code_Aster code-coupling. Second, it analyzes how such applications can be supported by XtreemOS and Diet. Third, it discusses how SALOME can make use of PadicoTM for enhancing network access.

2 Application classification

2.1 Definitions

Before analyzing the relationship between Programming Model Frameworks (PMFs) and Resource Management Systems (RMSs), let us define some vocabulary, aimed at disambiguating some commonly used concepts.

Application or scientific application is a set of executable and run-time libraries with the goal of performing a scientific computation. The data which is input to an application is transformed to output data which is useful to a scientist.

Task is an elementary execution unit of an application. Tasks can be either temporally or spatially related. A **temporal relation** means that a task needs to execute after another one completes, as the output of the latter is required for the computations of the former. A **spatial relation** means that two tasks need to execute at the same time, e.g., because they exchange data during their execution. For example, **data-parallel tasks** simultaneously execute the same code on different fragments of the input data.

Job is a resource allocation abstraction offered by RMSs. A job usually assigns minimum resource requirements to a piece of code that eventually gets executed. If the absence of additional QoS constraints (such as dead-lines), the allocation is served in a best-effort mode, once enough resources become available. It is the RMS that decides when the job is started.

Advance reservation (or reservation) is a common resource allocation abstraction offered by RMSs, in which the allocation is started at a fixed, pre-established time. Advance reservations can be seen as an agreement with somewhat stricter QoS guarantees.

These definitions are meant to clear up some possible confusions. First, we distinguish a task, which does not provide by itself any useful result, from an application, which is a set of tasks. Second, we distinguish between jobs and applications. An application may submit several jobs (or several reservations) to the RMS, depending on how it decided to present its resource requirements to the RMS, subject to the abstractions provided by the RMS.

Since in this deliverable we are analyzing the PMF—RMS interactions, we are interested in characterizing the resource requirements at the **application** level.

2.2 Application Resource Requirements

Let us analyze the different types of applications depending on their resource requirements. Figure 1 offers an overview of the properties we found relevant for our analysis, which we have grouped according to two criteria: **dynamicity** (i.e., variation in time) and **QoS** requirements.

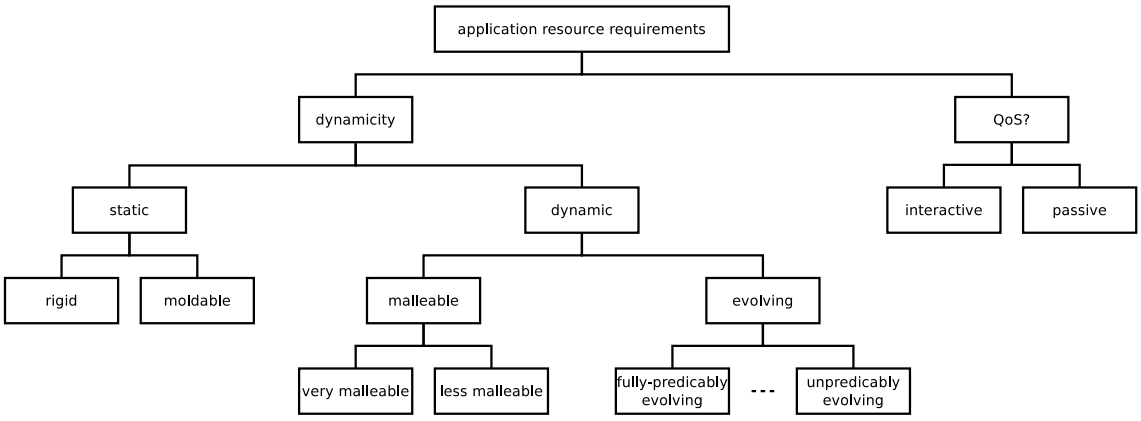

Figure 1: A taxonomy of application's resource requirements

2.2.1 Dynamicity

Let us characterize applications depending on how their resource allocation may vary in time. Applications can either be static or dynamic.

Static applications. The allocated resources of a static application are fixed at start-time and cannot change afterwards: the resources used by a static application are constant throughout its execution. A static application can either be **rigid**, i.e., it can only run on a hard-coded configuration of resources, or **moldable**, i.e., it can run on any configuration amongst a well-defined set of configurations of resources.

Note that, the terms rigid and moldable can have various shades. For example, an application that can run on exactly two hosts of cluster C seems to be "less rigid" than an application that can run only on hosts h_1 and h_2 of that cluster. Similarly, an application requiring all hosts to be on the same cluster seems to be "less moldable" than an application whose hosts can belong to multiple clusters.

Dynamic applications. As opposed to static applications, the allocated resources of a dynamic application can change throughout its execution. Unfortunately, the word dynamic is very vague, as it does not characterize *who* or *what* is determining the allocation to change. Therefore, we further characterize dynamic applications as being **malleable** and/or **evolving**.

Malleable applications. The allocated resources of a malleable application can change (grow/shrink) due to resource-related constraints. For example, if additional resources are available, an application might want to take advantage of them and speed-up its execution. Similarly, if resources are becoming scarce, an application might reduce its resource usage, so as to conform to a system policy. When the set of allocated resources changes, a malleable application has to adapt, which can be a more-or-less costly operation. Depending on the required bandwidth, time, CPU power and disk capacity wasted[1] during the adaptation process, we divide applications into i) **very malleable** applications that can quickly grow / shrink as required by the availability of the resources, and ii) **less malleable** applications that are heavily penalized (i.e., slowed down) for each adaptation.

To efficiently make use of resources, a malleable application requires that its allocated resources be within its minimum and maximum requirements. Such requirements can usually be determined at the start of the application. Minimum requirements are imposed by the complexity of the computations, e.g., the amount of memory needed to store the working set. Shrinking the application below this level forces it either to abort computation or to checkpoint and restart later. An application could be grown above its maximum requirements, however its current degree of parallelism would not allow it to make effective usage of these extra resources.

Evolving applications. The allocated resource of an evolving application changes due to the fact that the application's resource requirements change during its execution. This can either happen because the computation has grown or shrunken in complexity, or because of some external factors, e.g., a human operator changed some parameters of the computation. Depending on how much time in advance the evolution can be predicted (called the **horizon of prediction**), evolving applications can be grouped into fully-predictable, marginally-predictable and non-predictable. **Fully-predictably** evolving applications are those whose evolution can be fully described at the beginning of their execution, i.e., their horizon of prediction is infinite. **Marginally-predictably** evolving applications have a finite horizon of prediction, whereas **non-predictably** evolving applications change their requirements, without making it possible to have any predictions (i.e., the horizon of prediction is zero).

[1] We use the word "wasted" as we consider that adaptation is a secondary concern of the application, the primary one being to complete computation and output the result. In no way do we want to suggest that adaptation is not necessary, or that adaptation is hindering the application from executing efficiently.

As it has already been mentioned, applications can be both malleable and evolving. For example, either the minimum or the maximum requirements might change as a function of the application's internal state.

2.2.2 QoS

Depending on additional constraints that the application imposes on the time the resources are allocated to it, we distinguish two types of applications: interactive and passive.

Interactive applications. Such applications have strict QoS requirements, for example, because they need to cooperate with external entities, such as scientific instruments or human operators. These applications require the allocation to start at a precise moment of time. Once the allocation has started, these applications cannot tolerate a degradation of the allocation, e.g., suspending an application so as to resume it later is not an option.

Passive applications. In contrast, passive applications have more relaxed QoS requirements. They do not interact with external entities, therefore, the results they compute can be delivered some time later. The allocation of resources to these applications can be arbitrarily delayed. Also, during its execution, the application can be killed and restarted later, it can be suspended and resumed later, or it can be check-pointed and restarted. Of course, the allocation of resources has to be made carefully, so as to guaranteed that the application will eventually finish, most likely before a given deadline.

2.3 Example of Applications

This section gives some popular examples of applications and characterizes their resource requirements, in order to clarify the above taxonomy.

SPMD applications are generally moldable. For example, MPI applications usually use the `MPI_Comm_Size` function to adapt their execution to the number of processors they have been launched on. Nevertheless, MPI applications can be rigid, e.g., a "ping-pong" application might only compute correctly without crashing, if it runs on 2 processors. On the other hand, MPI-2 applications can be made malleable [3, 10] or evolving to a limited degree, using the `MPI_Comm_Spawn` function.

Bag of Tasks (BoT) and **Parameter-Sweep Applications (PSA)** are generally very malleable. They can adapt to a varying number of resources during their execution. Growing is handled by launching new tasks on the newly allocated resources, while shrinking is handled either using checkpoint-resume or kill-restart [9]. Since tasks are generally small, adaptation can be achieved with a low cost. These applications are generally not evolving, because the number of tasks to execute is fixed from the beginning.

Workflows are generally malleable and evolving. They are malleable because they can adapt to various resource configuration even at run-time; this is often a by-product of their fault-tolerance handling. They are also evolving as the workflows may have various branching factors, which increase/decrease the resource requirements. Regarding predictability, workflows may be fully-predictable if all the tasks are known at start-time (in literature these are commonly called **static workflows** [11]). However, some workflows create tasks depending on the output of previous tasks, in which case, the workflow is only marginally predictable.

Adaptive-Mesh Refinement simulations (AMR) are gaining increasing momentum. Since, the simulated phenomena are rather chaotic, AMR applications are necessarily non-predictably evolving. If implemented property, such application can be made malleable, with minimum requirements as low as to keep the refined mesh in memory, and with maximum requirements as high as to make the application execute efficiently (e.g., at their maximal speedup). Due to the computations, both of these requirements change during the computation. Unfortunately, as of today, most AMR applications are programmed so that the varying resource requirements are not externally exposed. They are treated as moldable applications [2], with the number of processors

empirically chosen, so as to most likely stay within the minimum and maximum requirements throughout the whole computation.

3 GridTLSE and Diet

GridTLSE is an expert system for solving large, sparse matrices using direct solvers [1]. At its core it uses a scenario language, which describes an abstract workflow. GridTLSE belongs to the family of **evolving applications** defined in Section 2. In order to launch the tasks on computing resources, GridTLSE relies on Diet.

Diet is a hierarchical middleware, aimed at easing the access to computational resources on large-scale platforms, such as Grids. Diet already offers a variety of services such as scalable, hierarchical scheduling of computation requests, data management and data-flow scheduling.

As a prerequisite to the COOP project, GridTLSE has to be modified to fully use the above features. Since a scenario is in essence a dynamic workflow, GridTLSE's engine unrolls the scenario to generate a data-flow. When a branch whose action depends on previous results is encountered, the so-far-obtained data-flow is submitted to Diet, which takes care of scheduling and executing it. After Diet has finished executing the data-flow, GridTLSE resumes unrolling the scenario and generates a new data-flow which again is submitted to Diet. The process continues until the whole scenario has been executed.

While this approach gives satisfactory results in simple cases, we have identified use-cases, in which GridTLSE needs to influence the decisions taken by Diet. We shall present two such examples: "threshold pivoting" and "multiple right-hand sides".

3.1 Threshold Pivoting

3.1.1 Problem Description

Direct methods are one type of methods to solve linear systems $Ax = b$. These methods usually consist of different steps: analyse, factorisation, solve. During the factorisation step of unsymmetric matrix, factors L and U are computed such as $L.U = A$.

During this factorisation, in order to preserve the numerical stability (i.e., avoiding the division by a small number) pivoting can take place. One way to select the pivot is to choose a diagonal entry according to a given threshold. The selection of the threshold is important and for some values, the result of the solution of the linear system can be very bad.

Depending on the available resources, we want to run as many factorizations as possible with different thresholds. This would allow us to find the best LU decomposition given a limited amount of time. However, choosing the number of requests to send to Diet would require to know the availability of the resources in advance.

3.1.2 Problem Analysis

The above problem assumes that the computations are composed of two parts. **Necessary computations** are those that need to be computed so as to satisfy the user's minimum precision requirements. **Optional computations** allow the user to have more precision if enough resources are available, so that the end-time would not be delayed. This is clearly a case of *malleability* (Section 2.2).

Returning to GridTLSE and Diet, our main issue is how should GridTLSE (by collaborating with Diet) choose the number of requests to generate. To tackle this problem two directions can be envisaged: "resources-full" feedback and intelligent resource selection.

"Resource-full" feedback In this approach, GridTLSE would send requests to Diet until the latter signals that all resources are occupied with requests. The purpose here is to send as many requests as possible without overloading the system. First, values that are of utter importance (e.g., 0, 0.5, 1, etc.) are sent, then the least important values are treated, so as to cover as much of the interval as possible. When previously submitted requests are done, if the dead-line is not exceeded, the generation of new requests can continue. If the dead-line is reached and all necessary computations are finished, extra requests should be cancelled.

Supporting such a solution requires relatively little information from the RMS. GridTLSE would only require a feedback which tells it whether it should continue generating requests or not (i.e., have all resources received at least a request?). On the other hand, GridTLSE would require very little information about its tasks. For example, there would be no need to model the duration of a request.

Despite its simplicity, such a solution might prove unsatisfactory. For example, there would be no way for GridTLSE to inform the user whether the dead-line is too strict. Also, if resources are heterogeneous, it would be preferable to send necessary computations to faster resources, so as to ensure that they finish as early as possible. Due to these limitations, we have thought of a second direction the solution might take.

Intelligent resource selection In this approach, an algorithm would have to be written. It would take as input application-specific information and resource-specific information, then it would compute the number of requests to send. If resources are heterogeneous, the algorithm would also compute on which resource a particular request should be sent to.

Let us delay the analysis of this approach until Section 3.2.2 as it is very similar to the second use-case.

3.2 Multiple Right-Hand Sides

3.2.1 Problem Description

When solving sparse equations, it is often the case that there are multiple right-hand side vectors. In other words, given a matrix A of size $m \times n$ and a matrix B of size $m \times p$, one wants to find the matrix X of size $n \times p$, such that $AX = B$.

In order to efficiently solve the problem, the matrix B can be split in matrices $B_1, B_2, \ldots$ having $p_1, p_2, \ldots$ columns ($\sum_i p_i = p$). A critical issue is how the numbers p_i should be chosen? If enough resources are available, it is preferable to split B in many matrices, so as to "create" enough parallelism and keep all resources busy. Most likely submitting single column matrices will give satisfactory results.

However, if few resources are available, the above approach gives poor performance. Due to the way the underlying dense matrix operations work, it is more efficient to solve 2 columns simultaneously, than 1 column at a time (Figure 2). Therefore, choosing a good decomposition should be done by taking the availability of the resources into account.

The problem becomes more difficult on heterogeneous platforms. Since each host has a different computation power, the time necessary for solving the same number of columns differs from one host to another. Thus, decomposing the right-hand side should take into consideration the relative computation power of the hosts.

If communication time is non-negligible, network-related information has to be considered. For example, if a host is connected through a slow network, assigning a sub-matrix to it might actually delay the end-time of the whole computation, due to the fact that transferring data requires a considerable amount of time. In this case, choosing a right-hand side decomposition would require information about the network topology, latencies and bandwidths.

3.2.2 Problem Analysis

In order for GridTLSE to take advantage of the above mentioned optimizations, an algorithm has to be devised. This algorithm would take as input both application-specific information (size of the matrices, post-analysis matrix information, ...) and resource-specific information (size of memory, CPU processing power, perhaps some microbench-marks, network topology, ...). The algorithm would output both the decomposition of matrix B and (in the case the resources are heterogeneous) the resource on which each sub-matrix should be solved.

Since this algorithm requires information from both the application and the RMS, a fundamental problem is which actor should execute it. Should the algorithm run on the RMS's side

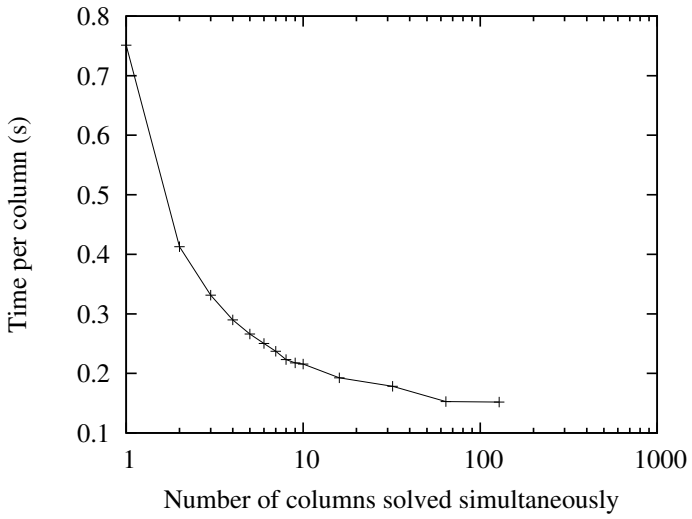

Figure 2: Performance for solving multiple right-hand sides.

requiring information to be retrieved from the application or shall the algorithm run on the application's side requiring information to be retrieved from the RMS? Once a choice has been made, the next question which need to be answered is: What interface should be put in place between the application and the RMS?

Currently, DIET has the possibility to use an application-specific scheduling algorithm. It allows a DIET client to choose which service daemon a request should be dispatched to. This approach assumes that the requests are known **before** the resource discovery phase. However, the above algorithm would require the requests to be generated **after** the resources are discovered. Therefore, a possible solution might be to extend DIET's client interface so as to allow an application to hook between the resource discovery and the request submission phases.

3.3 Contribution to the COOP Project

To pave the way for an efficient collaboration between GRIDTLSE and DIET, we propose ourselves to carry out the following steps.

First, the DIET architecture should be extended, so as to allow the client-side scheduler to take more advanced decisions. For example, it should be able to split requests depending on resource-specific information.

Second, since GRIDTLSE submits data-flows to DIET, an interesting direction would be to study how the generated data-flow should be optimized to the available resources. The issue is further complicated by the fact that, in order to do multi-workflow optimizations, a different agent (MADAG) is responsible for workflow scheduling.

Finally, in order to evaluate the above approaches, an algorithm would have to be devised for each of the two use-cases.

4 SALOME

SALOME is an open-source platform for numerical simulation. It provides a set of generic modules allowing to perform the main steps of a numerical study into the same platform: geometry, meshing, data definition, calculation and post-processing. Within the COOP project, it is the calculation module named YACS that is targeted. In order to use SALOME for building a numerical application (aka a calculation schema in cases targeted in COOP) two steps are needed. First, individual codes are wrapped into SALOME components, so as to ease reuse. Then, a schema is written which specifies how components are to be connected. It also specifies what are the spatial and temporal relations between the components. Finally, the YACS module executes the schema with the use of SALOME's services like ressources management or naming service.

The power of the SALOME platform lies within the fact that it allows components to be arbitrarily composed, both spatially and temporarily. Applications can range from being composed completely spatially (such as usual code coupling applications) to completely temporarily (such as usual data-flow / work-flow applications). Thus, SALOME applications can have an arbitrarily complex structure.

When running such applications on HPC resources one needs to select the set of resources which would enable an efficient execution. Given the diversity of available codes and composition structures, it is unlikely that a generic optimization algorithm could be devised, that would give satisfactory results for all SALOME applications. Therefore, we would like to enable each SALOME application to employ its own, specialized resource selection algorithm. We also want to enable communication between the schema executor of YACS and the resource management system to be able to address dynamic resources availability.

The rest of this section is organized as follows. First, two SALOME applications are introduced and analyzed. Next, the support of each RMS for these applications is studied and the challenges that appear are highlighted.

4.1 Use-cases

Let us highlight a few SALOME applications and focus on a few use-cases to extract their needs in terms of resource usage.

4.1.1 HLW Application

The HLW application (presented in [1]) mainly consists of two parallel for loops which launch data-parallel tasks. Two parameters determine the structure of the application:

- the **branching factor** which is the number of data-parallel tasks to unroll;
- the **cardinality** of the data-parallel tasks.

To efficiently execute the application both of these parameters need to be adapted to the available resources.

First, we shall consider the application **moldable** and study how to choose the application's parameters at launch time (i.e., static adaptation) on a single cluster. This would require the RMS to provide enough information so that the number of available resources could be determined.

Second, we shall build on the previous use-case and target a **multi-cluster platform**. The application is assumed to be single-cluster, i.e., it will never run on two or more clusters simultaneously. Thus, the problem is not only to select the number of hosts, but also on which cluster the application should execute. This would not only require more information from the RMS, such as the availability of each cluster, but also an interface to choose the target cluster.

Third, we shall enable a **multi-cluster execution** of the application. This would require the underlying RMS to be able to **co-allocate** resources on multiple clusters, i.e., enable the simultaneous availability of hosts belonging to different clusters.

In the above use-cases, we assumed the application is moldable. However, an improved resource usage would be enabled if the application were considered **malleable** and the branching factor is

changed during execution (i.e., dynamic adaptation). We shall leave this as a secondary target, since properly supporting applications with a dynamic structure in SALOME is still being studied [8].

4.1.2 Decomposition domain with Code_ASTER and SALOME

This application targets a new way of performing domain decomposition with Code_Aster. To be able to address complex problems, the domain of execution has to be divided in many parts to be able to be run in the calculation nodes. For each step of the calculation, each process of Code_Aster performs a calculation on its own domain and sends the results to a coordinator that performs a convergence computation. Then the coordinator sends data to the nodes and a new step is calculated. In COOP the coordinator and the algorithm are outside of Code_Aster. Basically, the application is composed of a coordinator component, many instances of Code_Aster (one instance for each domain).

Dependending of the size of the problem and the availability of the ressources, the number of domain could change. This application targets the same functionnality than the first use case, but it also needs high performance communication between Code_Aster nodes and the coordinator nodes. The coodinator could also be parallel, but this problem is not in the COOP scope.

4.2 SALOME on XtreemOS

XtreemOS is a grid operating system based on Linux. The main particularity of XtreemOS is that it provides for grids what a traditional operating system offers for a single computer: hardware transparency and secure resource sharing between different users. XtreemOS provides also a native interface to manage Virtual Organizations (VOs).

4.2.1 Running SALOME on XtreemOS: Initial situation

Since XtreemOS supports only rigid applications for now, a YACS schema can be run as a rigid application. Two possible ways to run a SALOME application on XtreemOS have been identified.

Generate the machine list from a reservation and intercept remote process creations The `rsh` command is symbolically linked to a `xsubsh` command (similar to `xsubMPI` command for MPI programs). That command will check the XtreemOS user credentials and the resource reservation time, and it will create and start a process related to the current job.

To run a YACS schema on XtreemOS, first a reservation must be performed. Then, the `CatalogResources.xml` file is generated from the reservation. Finally, the YACS executor is submitted as a XtreemOS job that will use the generated `CatalogResources.xml` file, and indirectly the command `xsubsh` when it creates a remote process.

This technique, that consists in producing the machine list from the reservation into a format usable by the existing system and by intercepting `rsh` calls, can be applied to most existing systems to run rigid applications.

Create a backend to LibBatch SALOME's `Launcher` module provides a job interface where the user can create and launch jobs. This module uses the SALOME's internal resource manager, but it can also use batch managers through the LibBatch library. The LibBatch interface is generic enough to be implemented by different existing batch managers. So, SALOME can be run on XtreemOS by creating a backend to LibBatch, and by setting the Launcher module to use LibBatch.

4.2.2 Contribution to the COOP project

Let us present the elements that need to be changed to run SALOME applications as autonomous and adaptable applications on XtreemOS as a batch manager that notifies resource changes.

Changes to perform in SALOME SALOME's resource manager should be extended to manage resource change notifications, query on resource proximity, etc.

Changes to perform in XtreemOS

Implement a batch mode A batch mode could be implemented based on the existing advance reservation mechanism. In this mode, jobs will have an exclusive access to resources; a reservation time will be required; processes will be killed if they are not finished at the end of their reservation; and when resources are freed before the end of their reservation, notifications will be sent to waiting jobs.

Implement resource change notifications Use signals or publish-subscribe system to notify jobs about some resource changes that they subscribe to.

Implement queries related to resource topology Extend the resource query interface to ask for "close" resources based on the Vivaldi algorithm [4].

4.3 SALOME on Diet

As a use case to find the elements of cooperation between RMSs and PMFs, we are interested in studying how Diet (already presented in Section 3) could efficiently support SALOME applications. This would allow us to reuse some features of Diet such as hierarchical scheduling and data management.

As a prerequisite to the COOP project, SALOME would have to be ported to Diet. The main question to answer is **what services the** SALOME **SeDs should provide.** They could either provide a simple "launch executable" service or they could provide access to SALOME entities, such as containers.

Let us first focus on the static adaptation of SALOME applications. For single cluster execution, one would have to decide **what information the SeDs should provide** in their estimation vectors and **how should the client take advantage of it**. For multi-cluster execution, Diet would require support for service **co-allocation**, so to say, the simultaneous execution of two services.

As for dynamically adapting the application to the target resources, Diet currently assumes a "pull" model, where the application asks for the state of the resources before sending a Diet request. There is currently no mechanism to inform the application that the resource state has changed. Therefore, to dynamically adapting applications in Diet, **support for notifications** is needed.

4.4 SALOME and PadicoTM

PadicoTM is a high-performance communication framework for grids. It is designed to enable various middleware systems (such as CORBA, MPI, SOAP, JVM, DSM, etc.) to utilize the networking resources found on grids. PadicoTM aims at decoupling middleware systems from the underlying networking technlogies to reach transparent portability and flexibility: the various middleware systems make use of PadicoTM through a seamless virtualization of networking resources; only PadicoTM itself uses directly the networks. Its architecture is based on software components. Puk (the PadicoTM micro-kernel) implements a light-weight high-performance component model that is used to build communication stacks.

PadicoTM uses a three-layer approach [6]: the lowest layer does multiplexing and arbitration between concurrent accesses to a given network, and between accesses to different networks (e.g. TCP/Ethernet and Myrinet) on the same machine; the middle layer is the abstraction layer, based on dynamically assembled components; the higher layer, or personality layer, adapts the API to the expectations of applications.

Since SALOME components may communicate through various protocols, PadicoTM could be used to offer a uniform abstraction for the network. Thus, SALOME applications could work even on complex networks (e.g., those that include firewalls), without changing the applications themselves.

4.4.1 Manual configuration on a rigid set of nodes

The middle layer of PadicoTM is based on dynamically assembled components [5]. Various components exist to handle various cases, so as to be able to run any middleware system on top of any networking technology. However, this component assembly step has to be configured by the user to choose which communication methods to use between each pair of hosts. Moreover PadicoTM needs a *control channel* that forms an overlay network spanning all involved processes. To bootstrap its initial connection, this control channel needs a rendez-vous node to be started, and connection information to the rendez-vous node must be given to other nodes.

These mechanisms are easy to automate, and are actually fully automated, in the case of a *static* application, where the set of nodes never changes and is known by all other nodes, and the connections are established at initialization time. This is typically the case for MPI applications. In this case, the MPI process with rank 0 plays the role of rendez-vous node, and all conections are static.

On another hand, SALOME is dynamic, with dynamically connected CORBA connections, on a set of nodes that may evolve. If the set of nodes is *rigid*, it is possible manually configure PadicoTM with hardwired component assembly rules and hardwired scripts to manage the rendez-vous node and control channels connection establishment. This approach allows to run SALOME on PadicoTM as a proof of concept, but is not satisfactory for production use.

4.4.2 Dynamic adaptation of communication methods

For applications such as SALOME using a dynamic set of nodes and establishing dynamic connections, hardwired scripts and rules are not enough to properly configure PadicoTM. In such a dynamic context, PadicoTM should be able to automatically use the most appropriate communication methods by itself.

We propose to extend the existing mechanisms used for static applications on flat networks such as MPI applications, to manage dynamic and heterogeneous network topologies. Such mechanisms utilize network topology information to compute the best suited component assembly used by PadicoTM for each connection. Therefore, PadicoTM needs to get a full representation of the underlying network topology. However, getting information about topology is not straightforward. Various sources of topology information exists, but none of them is complete, regarding neither the included nodes or networks, nor the informations about nodes or networks. The topology information sources are:

- auto-detection: most network information may be detected by a process running on a given host. However, these informations cannot be used to *bootstrap* the initial control channel connection. Moreover, some complex topologies involving firewalls and private non-routed IP networks are hard to reliably auto-detect.
- existings resources directories: Adage [7] already uses a resource description file, and SALOME has a resource catalog. However these directories do not include enough information about networks that PadicoTM may need to manage all cases. A collaboration is useful to add more information into these files, or at least have a common data model to enable conversion between formats.
- Grid API: grids (e.g. Grid'5000) have an API to list their computing resources. Information gathered from such API contains only information about nodes and networks inside the given grid.
- manual configuration from users: as a last resort, a topology description given manually by the user may be needed. However, we do not want users to describe all the nodes and networks — potentially thousands.

The variety of sources, given that each resource description has been designed for a different goal, causes some issues, among them: the need to *convert* from one source or model to another; the ability to *merge* information from various sources; the need to uniquely identify resources, so as to avoid duplicates when merging.

Once we have a precise and dynamically updated description of the topology, we may implement a shortest-path algorithm to find the best suited component assembly when establishing a new connection.

4.4.3 Contribution to the COOP project

The expected contributions of this use-case are:

- a model for network topologies with all the needed information to dynamically adapt the communication methods;
- a cooperation and convergence between various topology models of PadicoTM, Adage, SALOME;
- a cooperation with Grid'5000/Hemera to make the Grid'5000 API evolve to contain more network-related information;
- a performance evaluation between hosts of the network topology, given to schedulers in PMF;
- get SALOME work on full Infiniband clusters, with all connections using Infiniband;
- get SALOME work accross private clusters, with private IP addresses and NAT, behind firewalls;
- make PadicoTM dynamically adapt its communication methods to the topology, to ease its use by dynamic PMF and applications.

5 Conclusion

This reports has first presented a taxonomy for classifying applications with respect to their interaction with an RMS. Hence, applications can be static (being further divided into rigid and moldable) or dynamic (being further categorized into malleable and evolving).

Then, the report presented use-cases brought forward by the two main COOP PMFs (GridTLSE and SALOME) and analyzed how they could cooperate with the two main COOP RMSs (Diet and XtreemOS). Moreover, how PadicoTM could enhance network access within the SALOME framework has been considered.

For each considered use-case, a work-plan has been identified. The remaining of the COOP project will aim to implement them.

References

[1] Julien Bigot, Frédéric Camillo, Ronan Guivarch, Aurélie Hurault, Christian Pérez, and André Ribes. Programming model frameworks for distributed high performance computing. Technical Report D2.1, ANR COSINUS COOP, ANR-09-COSI-001, jan 2011.

[2] Greg L. Bryan, Tom Abel, and Michael L. Norman. Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: Resolving primordial star formation. In *Proceedings, SC2001*. ACM Press, 2001.

[3] Márcia Cristina Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe O. A. Navaux. Supporting MPI malleable applications upon the OAR resource manager. 2009.

[4] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, pages 15–26, 2004.

[5] Alexandre DENIS. Meta-communications in component-based communication frameworks for grids. *Cluster Computing*, 10(3):253–263, June 2007.

[6] Alexandre Denis, Christian Pérez, and Thierry Priol. Network communications in grid computing: At a crossroads between parallel and distributed worlds. In *18th International Parallel and Distributed Processing Symposium (IPDPS2004)*, page 95a, Santa Fe, New Mexico, April 2004. IEEE Computer Society.

[7] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, November 2005. Springer-Verlag.

[8] André Ribes, Christian Pérez, and Vincent Pichon. On the design of adaptive mesh refinement applications based on software components. In *GRID*, pages 383–386. IEEE, 2010.

[9] Ozan Sonmez, Bart Grundeken, Hashim Mohamed, Alexandru Iosup, and Dick Epema. Scheduling strategies for cycle scavenging in multicluster grid systems. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:12–19, 2009.

[10] Rajesh Sudarsan and Calvin J. Ribbens. Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. *CoRR*, abs/cs/0703137, 2007.

[11] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *CoRR*, abs/cs/0503025, 2005.