Federation and abstraction of heterogeneous global computing platforms with the YML framework

Laurent Choy[1], Olivier Delannoy[2], Nahid Emad[2] and Serge G. Petiton[3]
[1]*Center for Computational Sciences, University of Tsukuba, Japan*
choy@ccs.tsukuba.ac.jp
[2] *PRiSM Laboratory, University of Versailles, France*
{Olivier.Delannoy, Nahid.Emad}@prism.uvsq.fr
[3]*CNRS/LIFL, Lille University of Science and Technology, France*
serge.petiton@lifl.fr

Abstract

Global computing platforms have become popular tools for the resolution of large scale problems. They are often independent without any interoperability between each other. Therefore, clients are now asking for a better availability and scalability. In previous work, we presented the YML framework which was a first attempt to enable the development and deployment of applications on several global computing middleware. However, it suffered from scalability issues and a static approach.

In this paper, we highlight recent extensions of YML. We achieve a dynamic federation of computing middleware because YML is now able to manage at the run-time several middleware back-ends. Other improvements such as an OmniRPC back-end, a component binary cache mechanism, a new data management module, and a new data type supported by the YML front-end, yield to a much better scalability. We present the first evaluations of these extensions on a network of workstations and with a typical distributed sort application. Although the results show a significant overhead, they stress the benefits of binary caching and dynamic back-ends federation.

Keywords

YML, peer-to-peer computing, grid computing, workflow programming language

1. Introduction

1.1. General context

Many grid and peer-to-peer computing middleware solutions (later called "global computing" middleware) have been developed and are becoming stable. They harness available computing and storage resources in order to build large scale platforms. Then, they are used to solve huge applications and/or store large amount of data. Although computing platforms have become popular tools, the current challenging issues are a better scalability and a better availability. Indeed, a large number of independent platforms are currently managed by different global computing middleware products without any runtime interoperability.

1.2. Main objectives of the YML framework

The global computing community is investigating ways to tackle such issues. For instance the GridRPC API [7] aims to standardize RPC. Also, a consortium proposed the IMPI [4] protocol in order to reach the interoperability between several implementations of the MPI standard. The YML framework proposes a higher-level approach and is not limited to any specific communication paradigm.

The first objective is to provide to the end-user a transparent access to computing and storage resources. So, the client only focuses on the algorithmic details of his application and does not take care to lower-level software/hardware considerations. To reach this objective, YML is based on a middleware-independent and intuitive workflow programming language called YvetteML. The later orchestrates reusable and platform-independent computational services.

Middleware-specific considerations are handled by a dynamic pluggable back-end mechanism. With this mechanism, we can dynamically federate many global computing middleware and we achieve our second objective which is to ensure a constant resource availability.

1.3. Essential contributions

Previous published work related to YML [2] concerned the definition of the workflow programming language, the specifications of the component programming approach and the YML design. Since that time, YML has been significantly improved. In this paper, we present evolutions in the following areas: YvetteML and services definitions, data management, and interoperability between middleware.

First, we provide new front-end facilities which make easier to develop applications. The YvetteML language proposes a new data-type in order to manage a collection of generic elements. It is useful to manipulate a very large data set as a parameter of component. Then, the YML front-end is able to handle any kind of files as input/output arguments of the YvetteML workflow as long as the client provides import and export routines. Those routines [2] convert (and reversely) heterogeneous and non-native data types into data types that can be manipulated by YvetteML.

Second, two improvements contribute to a much better scalability of YML. A Data Repository service is in charge of storage management of input/output component parameters, component binaries. Next, the YML worker unit (see Section 2) uses a cache mechanism in order to drastically reduce the number of transfers of component binaries. The cache mechanism also contributes to aggregate more resources which are located within lower bandwidth network.

Finally, a major contribution is a Multi-back-end scheduler which assigns at the runtime a component to any kind of middleware-specific back-end. By combining this back-end scheduler and a large choice of middleware-specific back-ends, we provide better federation of global computing middleware and as a result, a better availability of computing resources. In addition, we have implemented a new back-end for the OmniRPC grid computing middleware [6].

1.4. Outline

The remaining of the paper is organized as follows. Section 2 gives an overview of YML. Next, Section 3 presents the three main improvements to YML. The notion of *collection* is presented in Subsection 3.1. Then, Subsections 3.2 and 3.3 respectively discuss the improvements of data management, and the aggregation of resources from multiple middleware. Section 4 shows preliminary evaluations with a typical application. Finally, we conclude and present research perspectives in Section 5.

2. Overview of YML

YML is a workflow environment dedicated to the execution of parallel and distributed applications on various middleware. It relies on the notion of components. Components allow interoperability, modularity, reusability. They are also used for the definition and implementation of computing tasks.

2.1 Workflow representation: graph description language

YML proposes an intuitive representation of a distributed and parallel application by means of a workflow. A workflow consists of a graph whose vertices are independent and communication-less computing tasks. The edges of the graph represent the precedence relationships between the tasks. The graph description language is called YvetteML[1]. A graph represents a control flow, and not a data flow, since a precedence does not necessary concern a data dependency. The main structures of YvetteML are: the services execution, the parallel sections, the sequential loops, the parallelized loops, the conditional branch and the event notification/reception.

2.2 Services

YML represents the notion of computing task by components, called services. The reusability is the main motivation. Besides, this representation helps to clearly separate computational blocks of his application and communication expressing dependences. Each computational task is described by an *abstract service* and is implemented in an *implementation service*. Services information is contained into two catalogs. A Development Catalog stores information used at the time of the application development. An Execution Catalog stores information used during the execution of the application.

2.3 The YML framework architecture

Figure 1 gives a simplified view of an YML framework. We notice four entities.

First, the client provides the computational components (abstract and implementation services) and the application graph expressing the control workflow.

Second, the YML workers are managed by any kind of middleware as long as a back-end enables to handle it.

Thirdly, we consider YML. It hides to the client the complexity and the heterogeneity of the computing platforms. YML is composed of a front-end layer and a back-end layer.

- Inside the front-end layer, the component generator checks the semantic validity of the tasks and stores them in the Development Catalog. There is no middleware consideration at this stage. The YML compiler

[1]http://yml.prism.uvsq.fr/

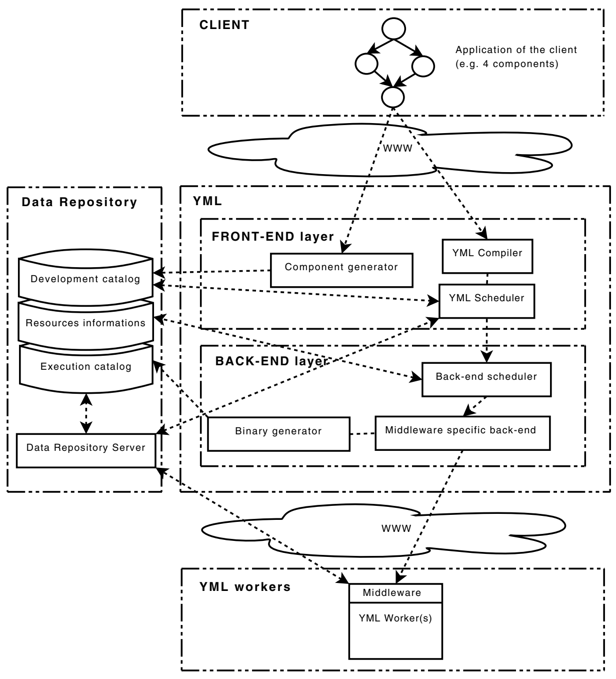

Figure 1. Overview of the YML framework

checks the validity of the graph and generates an intermediate representation which is handled by the scheduler. The intermediate representation is based on the event mechanism. An event can be assimilated as a boolean. A combination of events allows making pre- and post-conditions in order to formulate the dependencies between the tasks. The next task to run whose pre-conditions are validated is selected by the real-time YML scheduler and passed to the back-end layer.

- The back-end layer is roughly divided into two parts. First, the Back-end Scheduler schedules the current task to the appropriate back-end (see Section 3.3). The scheduling decision may rely on static and dynamic information extracted from a database: middleware information, network characteristics, statistics related to past executions, etc. Second, the middleware-specific back-end is in charge of submitting the task to the related middleware. The later assigns this task to an YML worker hosted by a computing resource. At this stage, the targeted middleware is known. YML can generate the suitable "binary services" to execute. This binary is stored in the Execution Catalog.

The fourth entity is the Data Repository server which interacts with the YML workers and the YML front-end (details in Section 3.2). Basically, this server is in charge of storing the binary of components and the input parameters. It delivers such data to the YML workers and collects output results which are used by the YML scheduler to select the next eligible task.

Finally, we stress the complementarity of three levels of scheduling. The first level is handled by the YML scheduler. It deals with dependences between computing services. The Multi-back-end scheduler is in charge of the second level. It performs the matchmaking between a service and a middleware-specific back-end. The last level of scheduling is not directly under the scope of YML: the computing middleware scheduler selects a suitable node to host an YML worker which executes the service.

3. Extensions to the YML framework

Work presented in [2] is a first attempt to use a workflow environment for the development and deployment of applications on several global computing middleware. However, it suffers from scalability issues when the size of the graph or the size of data increase. In this Section, we detail the contributions to solve such problems.

3.1. Collections

In the previous version of YML, the services could only manipulate a fixed number of parameters. It was not possible to set a variable number of parameters.

In addition, the YvetteML language does not provide any support for global operations. The end-users must implement them as a workflow. It is a complex process which leads to a large graph with lots of communication.

Therefore, we extended the YvetteML language and the component model of YML with the notion of collection. A collection is a sparse multi-dimensional array. Each element of the array is a piece of data. A collection can be manipulated as a scalar and passed to a service. The support for collection is present in the workflow description language and in services definitions. If a parameter of a service is defined as being a collection, the workflow engine will process the parameter differently so that the elements of the collection are made available to the service execution. The service can navigate the collection using iterators. Iterators allow fine grain control over the elements of the collection loaded in memory.

The notion of collection is middleware-independent and, it increases the scope of applications expressed by the workflow language and the services integration. Firstly, services can support variable or unknown number of parameters at the time of the creation of the service. If one of the service parameters is a collection then the service can either receive or create a large number of data. Secondly, using a collection of data is simple way to do out-of-core execution. The elements of a collection are loaded on demand from, and unloaded to, the persistent storage unit. Thirdly, collection provides an elegant solution to the execution of global communication. Finally, an appropriate use of collection can reduce the size of the graph and the number of communication, without adding any significant overhead.

3.2. Data management

YML data management relies on the notion of Data Repository. A Data Repository is used to represent the memory associated to an application. The memory is composed of resources identified by names in a way similar to Linda tuple spaces [3]. For the execution of a computational service, YML groups the input parameters in an archive and registers the archive to a Data Repository. When a service is instantiated on a middleware resource, YML executes a worker which acts as a container for the execution of the service. The worker interacts with the Data Repository in order to retrieve the service instance description, the archive as well as the implementation service executed on a peer of the middleware.

Repetitive transfers of the same implementation service generate a significant overhead. It is not possible to deploy in advance an implementation service on unknown peers, but we can transfer it only once for its first execution. This transparent cache mechanism is introduced in the YML workers in order to minimize the communication cost. It is currently only used for the implementation service but it could be generalized to every transferred data between the worker and the Data Repository. The cache mechanism of YML makes use of the available space in a predefined repository of the workers. It also provides a mechanism which guarantees that components are cleaned up after a fixed amount of time without being used. This is required in an environment such as YML where the computing resources are provided by volunteers.

3.3. Toward middleware globalization

One of the challenges for large distributed computing is the federation of resources under the control of a large variety of middleware. A solution to this problem can be proposed at several levels. The most widely adopted approach is at the middleware level. It consists in mapping the concepts of one middleware onto another by means of a "gateway". For example, a middleware deployed with Condor can be exploited through Globus using Condor-G [8]. A second approach federates several middleware at a higher level. This is the approach chosen by YML with the concept of back-ends in order to achieve middleware interoperability. A back-end allows YML exploiting any middleware. However, the model presented in [2] does not allow exploiting multiple middleware at the same time. Therefore we present an extension to the back-end layer which makes YML able to harness at the runtime several middleware. In the meantime the solution allows several YML workflow processes (i.e. applications) to be concurrently executed on the same group of middleware. Figure 2 illustrates the back-end extension. On the left side, it shows the initial model and, on the right side, it presents the new components which permit the globalization of resources.

The extension of the back-end model consists of three components: the *Multi-back-end back-end* which is similar to XtremWeb [1] or OmniRPC back-ends, the *Back-end Manager* and the *Back-end Connector*.

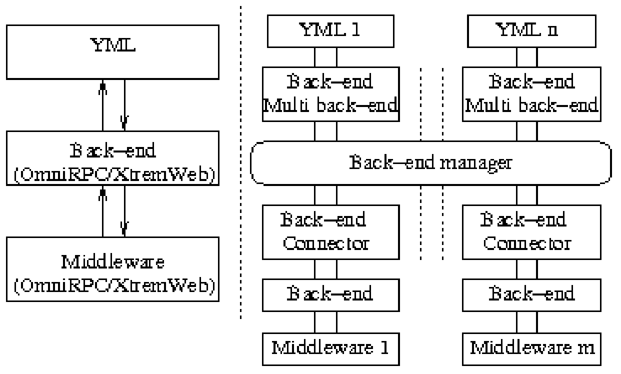

Figure 2. Globalization of computing resources with the back-end model extensions

Multi-back-end back-end component: YML interacts with the middleware through a back-end. This is unchanged by this extension. However, the so-called Multi-back-end back-end component does not communicate directly with middleware. Instead, it establishes a connection with the Back-end Manager via the network. This is the first step toward a distribution of YML.

Back-end Connector component: Regular back-end components, such as OmniRPC and XtremWeb ones, interact with middleware. This remains unchanged. They do not communicate with the Back-end Manager. We have introduced a component called Back-end Connector. This connector is used to relay information between the Back-end Manager and the regular back-end components.

Back-end Manager component: The Back-end Manager is the main component of this extension. It interacts with the two above components. It acts as a proxy between a set of YML workflow processes to be executed and a set of middleware providing computing resources. The Back-end Manager defines the notion of actors. There are two kinds of actors: workflow processes and middleware. They interact between each other through a component called a

Back-end Scheduler which is internal to the Back-end Manager. The Back-end Scheduler dynamically schedules work to suitable middleware. Scheduling choices are settled by means of information such as history of previous task executions, middleware statistics and workload.

The Back-end Scheduler component opens future research in multi-middleware scheduling and allocation strategies. Many strategies can be defined and enforce some of the following criteria: fault-tolerance, quality of services, computing capacities, software availability, load-balancing.

4. Evaluations of YML

4.1. DSORT: distributed sort application

We use a common case study application which consists in distributing and sorting an array of integers. It relies on a *Merge* step which takes 2 sets of integers S_1, S_2 and delivers 2 sorted sets S'_1, S'_2 where $card(S_1) = card(S'_1)$, $card(S_2) = card(S'_2)$ and $\forall x_1 \in S'_1$, $\forall x_2 \in S'_2$, $x_1 < x_2$.

4.2. Experimental platform

We harness a Network Of Workstations (NOWs) at the USTL (University of Lille 1), France. This NOWs is composed of a hundred of heterogeneous nodes which are non-dedicated because they are shared with students. CPUs varies from Intel Celeron 1.4Hz or AMD Duron 750MHz to Intel P4 3.2GHz. Main memory varies from 128MB to 1GB. The network bandwidth varies from 10 to $100 Mb/s$.

4.3. Evaluation of the component binary cache

Objective This first evaluation aims to show the benefits of the cache extension on the YML worker module. The cache mechanism is evaluated with the OmniRPC back-end.

Results Table 1 presents the wall-clock times needed to sort the arrays of integers. The DSORT application is using either 32, 64 or 128 blocks. Each block has either 10^3, 10^4 or 10^5 elements. Most of workstations are involved.

We compare the tests with and without the cache mechanism for the OmniRPC back-end. The decrease of the time-to-solution with the cache mechanism is significant because each service is deployed only once instead of one deployment per service execution. However, when the size of blocks is increasing a lot, the gain is decreasing because the amount of cached data on the YML worker (implementation services) is becoming small compared to the amount of transferred parameters (blocks). The cache mechanism is a promising way to reduce the time-to-solution and we plan to extend it to service parameters in the future.

	Size of blocks		
blocks/tasks	10^3	10^4	10^5
OmniRPC back-end, without cache on the worker			
32/560	00:08:48	00:09:46	00:20:04
64/2144	00:34:16	00:38:11	00:47:10
128/8384	00:50:38	00:52:10	00:54:18
OmniRPC back-end, with cache on the worker			
32/560	00:00:40	00:01:10	00:11:23
64/2144	00:01:00	00:05:08	00:47:55
128/8384	00:02:33	00:20:12	00:54:19

Table 1. Wall-clock time (in HH:MM:SS) of DSORT resolution at the USTL with the OmniRPC back-end

4.4. Evaluation with the multiple back-ends

Objective The aim of this second experimental step is to illustrate the performance of the multiple back-end mechanism. We propose various combinations of back-ends using the middleware currently supported by YML: OmniRPC and XtremWeb. Each back-end manages 22 workstations.

YML without Multi-back-end mechanism			
blocks/tasks	Omni b-e	XW b-e	
32/560	00:11:23	00:39:35	
64/2144	00:47:55	02:05:12	
YML with Multi-back-end mechanism			
blocks/tasks	1 Omni b-e	2 Omni b-e	3 Omni b-e
32/560	00:30:28	00:30:11	00:30:06
64/2144	01:54:21	01:51:47	01:49:35
blocks/tasks	1 Omni b-e and 1 XW b-e		
32/560	00:33:41		
64/2144	01:47:12		
Omni = OmniRPC, XW = XtremWeb, b-e = back-end			

Table 2. Wall-clock time (in HH:MM:SS) of DSORT resolution at the USTL with the Multi-back-end mechanism for *Size of blocks*=10^5

Results Table 2 presents the wall-clock times of six sets of experiments. The distributed sort application is using 32 or 64 blocks of 10^5 elements each.

Let us consider the experiments with the OmniRPC back-end(s). When we do not use the Multi-back-end version of YML, the time-to-solution is much shorter than with the Multi-back-end mechanism. That new facility is a significant source of overhead which can be compared to the overhead of the high-level XtremWeb [5] middleware versus the low-level and effective OmniRPC software. Indeed, the support for multiple back-end allows multiplexing computing resources between several applications (not presented in current experiments). It also provides fault tolerance because a back-end can disappear at any time without impacting the application execution. The execution continues either using the other available resources or waiting until a new back-end connects.

Next, without the Multi-back-end mechanism and by using the XtremWeb back-end, the time-to-solution is as long as all tests with the Multi-back-end facility. It means that the overhead of XtremWeb almost hides the Multi-back-end mechanism one.

Finally, the experiments underline that a unique YML application can be executed on two different middleware platforms based on OmniRPC and XtremWeb although such software are very different. More generally, YvetteML workflows can now cross the boundaries of middleware heterogeneity. On this point, we have validated the approach used in YML as a solution to the interoperability of middleware. The current limitation is the number of available back-ends. A Condor back-end will be released soon and Globus-based platforms are going to be tackled through the OmniRPC gateway to the GRAM resource broker.

5. Conclusion and perspectives

We have detailed three extensions to the YML framework. The notion of collection extends the YvetteML language and the service integration in order to support global operations and variable number of parameters per service. It also provides an out-of-core programming feature. The transparent data management has been extended in order to propose an automatic cache mechanism. It decreases the cost of communication and the load on the Data Repository servers. Finally, we have presented a solution to aggregate resources from different middleware at the same time. Now, an application can cross the boundaries of middleware heterogeneity and it benefits from a better availability and more idle resources managed by different systems.

We presented some preliminary results using a real experimental network of workstations at the USTL, France. The results showed a significant overhead when using the support for multiple back-ends. We explain this loss in performance by the structure of the current Back-end Manager which is similar to the XtremWeb dispatcher structure. It behaves as a proxy between the application and the resources. Therefore, it introduces a fault tolerance factor which may compensate moderate overhead.

We bring out three tracks for future work. First, the support for multiple back-ends adds a new level of scheduling. We expect that research on a more effective back-end scheduling policy will yield to a significant reduction of the time-to-solution. Moreover, this scheduling policy must take into account the data management aspects of the execution of an application. Second, the Data Repository system is minimal and needs to be improved. In spite of the component approach of the YML architecture, such work may impact several components of the current framework. Finally, future work will consist in the development of middleware-specific back-ends as explained in Subsection 4.4. The support of the DRMAA[2] specification is also considered.

6. Acknowledgment

We acknowledge the University of Versailles, the USTL and the INRIA Futurs, which have supported our work. Moreover, we sincerely thank the University of Tsukuba and the Polytech'Lille engineer school for allowing us harnessing computing resources.

References

[1] F. Capello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Neri, and O. Lodygensky. Computing on large scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Science*, 21(3):417–437, 2005.

[2] O. Delannoy, N. Emad, and S. Petiton. Workflow global computing with yml. In *Proceedings of the* 7^{th} *IEEE/ACM International Conference on Grid Computing*, pages 25–32, Barcelona, Spain, 2006.

[3] A. Deshpande and M. Schultz. Efficient parallel programming with Linda. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 238–244, Los Alamitos, USA, 1992.

[4] W. George, J. Hagedorn, and J. Devaney. IMPI: Making MPI Interoperable. *Journal of Research of the National Institute of Standards and Technology*, 105(3), 2000.

[5] C. Germain, V. Néri, G. Fedak, and F. Cappello. XtremWeb: building an experimental platform for global computing. In S. LNCS, editor, *Proceedings of the* 1^{st} *IEEE/ACM Internationnal Workshop on Grid Computing*, volume 1971, 2000.

[6] M. Sato, T. Boku, and D. Takahashi. OmniRPC: a grid RPC system for parallel programming in cluster and grid environment. In *Proceedings of the* 3^{td} *IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 206–213, Tokyo, Japan, 2003.

[7] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A remote procedure call API for grid computing. In *Proceedings of the* 3^{td} *International Workshop on Grid Computing*, 2002.

[8] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In *Global Grid Computing: Making the Global Infrastructure a Reality*, NJ, USA, 2003. John Wiley & Sons.

[2]http://www.drmaa.org/