Evaluating a data removal strategy for grid environments using colored Petri nets

Citation for published version (APA):

Trcka, N., Aalst, van der, W. M. P., Bratosin, C. C., & Sidorova, N. (2008). *Evaluating a data removal strategy for grid environments using colored Petri nets*. (Computer science reports; Vol. 0832). Technische Universiteit Eindhoven.

Document status and date:

Published: 01/01/2008

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Evaluating a Data Removal Strategy for Grid Environments Using Colored Petri Nets

Nikola Trčka, Wil van der Aalst, Carmen Bratosin, and Natalia Sidorova

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{n.trcka, w.m.p.v.d.aalst, c.c.bratosin, n.sidorova}@tue.nl

Abstract. In this paper we use colored Petri nets (CPNs) and the supporting CPN Tools for the modeling and performance analysis of grid architectures. The notation of Petri nets is a well known graphical formalism, able to model concurrency and different types of communication. CPNs extend Petri nets with timing, hierarchy, data, and programming language constructs, which makes them suitable for the modeling of grids. We use our grid model to evaluate a strategy for the optimization of data usage in grid environments. Our strategy is based on the automatic addition of clean-up tasks to grid workflows. By means of simulation we show that this strategy significantly reduces the amount of storage space needed to execute a grid application.

1 Introduction

Over the past few years grid computing has emerged as a powerful platform for executing data and computation intensive applications. Several grid architectures have been proposed to couple and orchestrate the available resources (e.g. [1, 9, 2]). However, as the variety and complexity of grid applications continuously increases, there is always a need for new solutions. Questions like “What is the best scheduling policy for my application?”, “Is it better to have a centralized or a decentralized information service?”, “When shall data be removed/replicated?”, etc. pop-up very often. To answer these questions one can perform experiments on an existing real grid platform. These experiments are trustful, but are also time-intensive, uncontrollable, limited to a small set of tests, and may disturb other (real) users. One way to avoid the problem is to build a simulation model of the grid. In this paper we propose Colored Petri nets (CPNs) [10] for the modeling and simulation of grid architectures.

Since grid simulation has been an active area of research for quite some time, there already exist a vast number of efficient and realistic grid simulators [11, 7, 8]. To show the reader that we are not building yet another

grid simulator, we list the reasons for using CPNs to model and analyze grid architectures: 1) CPN models are graphical, hierarchical, and have a formal semantics. A CPN model of a grid architecture can serve as an unambiguous descriptive model, clearly showing how different parts are internally designed and structured. CPN models are executable, so they can also model dynamic aspects of the grid, contributing to a better understanding of the whole mechanism. 2) The CPN language is supported by CPN tools [10], a powerful framework for modeling, functional verification (several techniques) and performance analysis (discrete-event simulation). The tool is especially suitable for performance evaluation as it provides monitoring facilities used to extract data during simulation, fully decoupling analysis from modeling. 3) (Coloured) Petri nets have been extensively used for many years to model different types of concurrent systems. There exists a plethora of models, from various fields, specifying concepts similar to those present in the grid [14]. Reusing these models, and/or the underlying ideas, makes the modeling of the grid in terms of CPNs a relatively easy task. 4) Most graphical grid workflow languages are either Petri-net based, or can easily be converted to Petri nets, like, e.g., those based on directed acyclic graphs. CPNs can, therefore, directly represent the user layer of many existing grid environments. They can be used as a testing framework for improving old and designing new grid workflow languages, or as a workflow language themselves, supporting all the patterns from [4, 15].

Several different types of grids exist (data grids, computational grids, etc.), and several ways of building a grid infrastructure have been proposed. Covering all of them in one single model would be a very difficult task. For this paper we restrict ourselves to a particular, albeit reasonably generic, grid architecture. Our architecture is suitable for specifying and executing grid applications that are computationally intensive, and in addition manipulate large sets of data. It can be seen as a computational grid in which data also plays an important role.

Our intention is not to model an existing grid architecture, but rather to provide a clear reference model for all the functionalities of a grid environment. We, however, believe that our model covers all aspects important for the analysis with enough detail, and that it can be used to discover trends that remain valid in more complex settings. The model is built in a highly modular fashion to allow for easy extensions.

A typical grid workflow is composed of several interdependent tasks. Every task usually specifies the set of the required input data files and the files that it generates. We frequently see examples of some data being used

only at one place in a workflow and never again. Although these data are not needed after a certain point in time, they typically stay on the grid until the complete workflow is finished. In a grid environment with reliable resources this is a waste of storage space, as some large data can remain stored for a very long time. To solve this problem we introduce a strategy that reduces the workflow data usage by automatically adding data clean-up tasks at the points from which no further tasks will access this data. The major challenge is to identify these points in a workflow language that allows for many complex routings. We use our CPN model of a grid architecture to show that the amount of storage space an application uses during execution is significantly reduced when our strategy is applied.

The rest of the paper is organized as follows. In Section 2 we define a grid architecture and present its CPN model. The proposed strategy and its evaluation are presented in Section 3. In Section 4 we give some conclusions, discuss related work, and provide directions for future work.

2 Grid Architecture

In this section we introduce a grid architecture and give the corresponding CPN model.¹ Only the main parts of the model are shown; the complete model can be obtained at www.win.tue.nl/~ntrcka/grid-model.

Fig. 1. Grid architecture - Top level

Our grid architecture consists of three layers. On the top we have the *application layer*, an environment in which the users describe their applications. On the bottom we have the *fabric layer*, the actual hardware of the grid. In between we have the *middleware layer*, responsible for scheduling, allocation of resources, and data handling. Figure 1 shows

¹ We assume that the reader is familiar with the CPN formalism; if not, [10] gives a good introduction

the main page of the CPN model. It consists of three sub-modules, each modeling one of the layers described above. The interface places model communication messages that are exchanged between the modules. We briefly explain these messages and the grid dynamics.

The grid fabric consists of nodes connected in a network. Every node can execute a set of services. The application layer issues a request to the middleware to run a particular service on the grid. It waits until it receives the corresponding (unique) id back, denoting the completion of the request. To look for a node that can execute this request, the middleware checks the current state of the nodes and the network. When a suitable node is found, the request description is sent to it, appended with the list of data that have to be transferred from other nodes (the request becomes an activity). The fabric layer reports when the activity is finished. We now explain each layer in more detail.

Application layer User applications specify the “flow of work” and are termed (*grid*) *workflows*. There may be different workflows using the same infrastructure, and there may be multiple instances of the same workflow, referred to as *cases*. Every workflow is a set of (interdependent) *tasks*. A *request* is a task instantiated for a particular case. It is an atomic piece of work that can be scheduled to run on the grid.

There are five different types of tasks: *stage-in*, *stage-out*, *computation*, *data removal*, and *routing* tasks. The first four we call *grid* tasks. Stage-in tasks are used for putting (local) data onto the grid. Similarly, stage-out tasks are for retrieving data from the grid. Data removal tasks are used to instruct the middleware to delete some data. These tasks are not expected to be specified by the user, but rather automatically added by a workflow preprocessor. Computation tasks are the real grid tasks, used for invoking a particular service on the grid. The routing tasks are used for routing purposes and to change parameters in other grid tasks.

All grid tasks are parameterized with a list of input data. Stage-in tasks must also specify the size of this data. A data element is represented by its logical name, leaving the user a possibility to parameterize the workflow with actual data. The data names are visible only to one instance of a workflow, i.e., they follow the so-called *case data pattern* [15]. This allows the user to run multiple instances of the same workflow at once, but with different physical data. Computation tasks also include the information on which service to call, and the list of data names for the generated output.

Every grid task can specify a node on which its requests are to be served. If this tag is set to 0 (“no node”), then the middleware decides for the best node. For data removal tasks, the value 0 for this tag means that the data needs to be deleted from all nodes.

The scheduling process in the middleware relies on estimates, so every computational task provides the following information: 1) the expected size of the output data when the input data size is 1000 (reference size), and 2) the execution cost on a reference node (a node with the minimal computing cost) when the input file has size 1000.

In this paper we focus on *process mining* [5] as a grid application. Process mining is a method for automatic discovery of process models (such as Petri nets) from event logs. As these logs are usually large files, and mining techniques are computationally intensive and decomposable, process mining is an ideal application for the grid. Figure 2 shows an application layer consisting of one workflow (multiple cases can be generated by the *GenCases* module) describing a typical process mining experiment. The event log file of size 1000, given by its logical name *Log*, is first sent to the grid. In the preprocessing phase of the mining process the log is filtered, and the result is stored in *FLog*. The next phase is the actual mining, i.e. process discovery, where a Petri net (*PN*) is obtained from the filtered log. This Petri net is checked for conformance with *FLog*, to see, e.g. how many traces from the log can be reproduced by the mined model. The result and the net are sent to the user, and the removal of all data takes place. The user layer is not informed when a delete request is finished. This is because the removal of data is not considered to be a real request but rather a signal to the middleware that some data is no longer needed. All tasks require from the middleware to find the most suitable nodes for their execution.

Filtering tasks are not computationally intensive and typically give output files that are slightly smaller than the input logs. Mining generates medium sized files but it performs heavy calculations. Conformance checking is of medium complexity and results in a very small file. These facts are reflected in the estimation parameters of the tasks in Figure 2.

Middleware The middleware layer is shown in Figure 3. Its main purpose is to allocate nodes to the requests arriving from the application layer. When a request is assigned to a node, it is called a *job*. Since the allocated node might not have all the required data, the middleware builds a list of data that needs to be collected from other nodes, called the *transfer list*. A job with a transfer list is called an *activity*. A *node schedule*, that the

Fig. 2. Application layer - A process mining experiment

middleware sends to the node, is an (ordered) list of activities that need to be performed on that node.

The arriving requests are put in the request queue. The complete queue is periodically taken by the *Scheduler*, turned into a set of node schedules (one for each node) and sent to the nodes. The middleware keeps the information of all scheduled jobs, i.e. on all the activities present in the system. When an activity finishes, this database is updated and a message is sent to the application layer (if the activity is not of the data removal type). To make scheduling decisions the middleware queries an information service for the current status of the nodes and the network.

Fig. 3. Middleware

Figure 4 shows the scheduling module in detail. As the request queue can be very large, the first step in the scheduling process is to select the number of requests to be considered in the current scheduling event (given as a parameter). The second step is to obtain the status of the

nodes and the network. This information comes in terms of the load of each node and the transfer costs for each network link. After the information is gathered, the actual scheduling takes place (any algorithm can be added in a modular fashion). Requests that cannot be scheduled in the current scheduling event are returned back to the queue and the whole process repeats after the given period (also a parameter). Our scheduling mechanism is fully adaptive, i.e. after every scheduling event it is possible to change all scheduling parameters.

Fig. 4. Scheduler

Fabric layer The grid is a set of *nodes* connecting in a *network*. Every node consists of a *computing element (CE)*, and of a limited *storage area*. A CE is characterized by its computational cost (can be seen as a combination of its speed and the current load); a storage area is characterized by its size. Every node holds the knowledge of the *services* that it provides, and the (grid) data that it stores.

We assume that storage space is fixed, but we allow computational costs to change dynamically. This is a realistic scenario in which the users outside the grid also use the nodes, reducing the available CE power, but grid applications have a certain amount of storage guaranteed.

The network is modeled as a set of *links*, where each link is characterized by the cost of sending data (of size 1000) through it. This cost can change over time and it represents an abstraction of the network's bandwidth and latency.

Figure 5 shows the main page of the fabric layer. Every node has a queue of activities that are scheduled to be performed on this node. These activities are taken from the queue in a FIFO fashion. Data removal activities simply remove the data from the allocated node. For the other activities, the data transfer takes place if the node does not have all

sound workflow net [3], i.e., must have a start and an end transition, always an option to complete, and no dead transitions.² Figure 6 shows an example of such a workflow (ignore the gray elements for the moment).

The algorithm works on a per data element basis. The first step is to find all markings (distribution of tokens over places) from which no transition labeled with the corresponding data element can be reached. This is done by inspecting the (finite) reachability graph of the workflow, assuming one token in the initial place. For the data element c in our example two such markings are the marking with one token in p_5 and one in p_6 , and the marking with one token in p_7 . For the data element d one such marking is the marking with one token in p_8 . The second step is to reduce the obtained set of markings by recursively eliminating those markings that can be reached only by passing through some other marking from the set. If two markings can reach one another, they are both left in the set. For the data element c the marking with one token in p_7 is eliminated; for the data element d , however, both the marking with one token in p_8 and the marking with one token in p_9 are kept. In the next step of the algorithm a data removal task is introduced (parameterized with the data element in consideration) for each marking remained in the final set. If a place contains $n > 0$ tokens in this marking, an input/output arc from this place is added to the new task, with arc weight n (not shown if 1, as is always the case in our example). This ensures that a removal request can be issued when the net is in this marking. Finally, a shared input place is added to the introduced tasks, containing one token (for each case), to make sure that every removal request is issued only once. These additions are illustrated in gray color in Figure 6. To make sure that data removal task are not skipped, they have priority over other tasks (not shown in the figure).

Fig. 6. Data clean-up tasks added to a workflow

Evaluation Our testbed is as follows. Every node has the storage area of size 5000, 7000, or 10000. The cost for calculation is 1, 5, or 10. We

² Soundness should not be seen as a restriction but rather as a sanity check for the workflow.

ran experiments on a grid that has 9 nodes, one for each combination of the above numbers. The cost of data transfer between any two nodes is assumed to be 200.

We use the Min-Min scheduling algorithm [12], modified to take the data transfer times into account (similarly to [6]). We do not use forecasting but base scheduling decisions solely on the current node and network status. The running times of tasks and the output data size are always correctly estimated. It is assumed that scheduling time is insignificant compared to the other delays in the system.

The evaluation is done under the assumption that the input application is the process mining experiment from Figure 2. One difference is that we do not take every log to be of size 1000, but for each case take a uniformly distributed sample between 500 and 1500 instead.

The performance measure in question is examined for different case arrival rates, ranging from 1/175 to 1/125. The scheduling period is also varied from 0 to 100. We perform 10 independent simulations for each of the examined configurations, and we calculate 95% confidence intervals. Each simulation run is limited to 1000 cases.

Table 1 shows the results of the simulations. We show there what percentage of storage space used when our strategy is not applied becomes available when the strategy is applied. We see that the improvement is greater for longer scheduling periods, and that it decreases when cases arrive less often. This is expected as longer scheduling period produce better schedules, and less cases implies that less storage space is needed.

4 Conclusion

We modeled a grid architecture in terms of Colored Petri nets. Our model is formal, graphical, and executable, offering a clear and unambiguous view how different parts of the grid are structured and how they interact. It covers all features important for performance analysis, it is fully adaptable and easily extendible.

To fight the problem of some data occupying the grid storage space unnecessarily long, we introduced a method for the automatic addition of data clean-up tasks to grid workflows. We evaluated the method by conducting several simulation experiments using our model of the grid. The results showed that under a heavy load of process mining applications, our method could reduce the required storage space by as far as 80%.

Related work To the best of our knowledge, only one attempt has been made to formally specify the semantics of grids. In [13] a semantical model

Table 1. Improvements in data usage

for a grid system is given using Abstract State Machines as the underlying formalism. The model is very high level (a refinement method is only informally proposed) and no analysis is performed.

In order to analyze grid behavior several researchers have developed grid simulators [11, 7, 8]. These simulators are typically Java or C implementations, built for the analysis of scheduling and data replication algorithms. They do not provide a clear reference model as their functionality is hidden in code. It is difficult to check the alignment between the real grid and the simulated grid.

Automatic addition of clean-up tasks is standard practice in the Pegasus grid environment [9]. Their workflow language is based on direct acyclic graphs while our method works for the more general language of Petri nets. They perform static scheduling and their algorithm works on a per node basis. As we work in a dynamic environment, we need to add clean-up tasks to delete data from *all* nodes.

Future work The strategy we introduced in this paper is static, being based on workflow preprocessing. For future work we plan to develop and evaluate more dynamic approaches in which the clean-up functionality is built directly into the scheduling algorithm.

The algorithm presented in this paper is based on the analysis of the reachability graph of a workflow net and is thus of high complexity. We will identify subclasses of Petri nets for which the analysis can be done directly on the Petri net.

References

1. Condor Project. <http://www.cs.wisc.edu/condor/>.
2. K-WfGrid web site. <http://www.kwfggrid.eu>.
3. W.M.P. van der Aalst. Verification of Workflow Nets. In *Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 407–426. Springer-Verlag, 1997.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
5. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
6. A.H. Alhusaini, V.K. Prasanna, and C.S. Raghavendra. A Unified Resource Scheduling Framework for Heterogeneous Computing Environments. In *8th Heterogeneous Computing Workshop (HCV'99)*, pages 156–170, 1999.
7. R. Buyya and M.M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002.
8. D.G. Cameron, A.P. Millar, C. Nicholson, R. Carvajal-Schiaffino, K. Stockinger, and F. Zini. Analysis of Scheduling and Replica Optimisation Strategies for Data Grids Using OptorSim. *J. Grid Comput.*, 2(1):57–69, 2004.
9. E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.
10. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
11. A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *3rd International Symposium on Cluster Computing and the Grid (CCGRID '03)*, pages 138–146, Washington, 2003. IEEE.
12. M. Maheswaran, S. Ali, H.J. Siegel, D.A. Hensgen, and R.F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCV'99)*, pages 30–44, 1999.
13. Z. Németh and V. Sunderam. Characterizing Grids: Attributes, Definitions, and Formalisms. *J. Grid Comput.*, 1(1):9–23, 2003.
14. W. Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer, 1998.
15. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *LNCS*, pages 353–368. Springer-Verlag, 2005.