A Reference Model for Grid Architectures and Its Analysis

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

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

Abstract. Computing and data intensive applications in physics, medicine, biology, graphics, and business intelligence require large and distributed infrastructures to address today's and tomorrow's challenges. For example, process mining applications are faced with terrabytes of event data and computationally expensive algorithms. Increasingly, computer grids are used to deal with such challenges. However, despite the availability of many software packages for grid applications, a good conceptual model of the grid is missing. Grid computing is often approached in an ad-hoc and engineering-like manner. This paper provides formal description of the grid in terms of a colored Petri net (CPN). The CPN can be seen as a reference model for grids and clarifies the basic concepts at a conceptual level. Moreover, the CPN allows for various kinds of analysis ranging from verification to performance analysis. In this paper, we show that our reference model allows for the analysis of various distribution strategies using simulation.

Keywords: Computational grids; grid architecture; colored Petri nets.

1 Introduction

Developments in information technology offer solutions to many complex problems, but they also lead to new challenges. The idea of collaboration and distribution of work in order to solve a given problem is promising but complicates matters dramatically. Ideas like distributed computing or service oriented architectures have been embraced by the scientific and industrial communities. *Grid computing* uses available technologies to approach distributed computing resources linked via networks as one computer.

Despite the availability of a wide variety of grid products, there is little consensus on the definition of a grid and its architecture. In the last decade many researchers tried to define what a grid is. Some argue that grid computing is just another name for distributed computing, while others claim that it is a completely new way of computing. Recently, in [17], the author presented the outcome of a survey conducted among grid researchers all over the globe. The

R. Meersman and Z. Tari (Eds.): OTM 2008, Part I, LNCS 5331, pp. 898–913, 2008.

main conclusion, on which the most of the respondents agree, is that grid computing is about sharing resources in a distributed environment. This definition, however, only offers an idea of what a grid is and not how it is actually working.

In order to classify all the functionalities that a grid system should provide, [13] describes a grid architecture as composed of five layers: (1) *fabric*, providing resources such as computational units and network resources; (2) *connectivity layer* composed of communication and authentication protocols; (3) *resource layer* implementing negotiation, monitoring, accounting, and payment for individual resources; (4) *collective layer* focusing on global resource management; and finally, (5) the layer composed of user *applications*. Similar classification is given in [1] where the architecture is composed of four layers: (1) *resources*, composed of the actual grid resources like computers and storage facilities; (2) *network*, connecting the resources; (3) *middleware layer*, equivalent to the collective layer of [13], but also including some of the functionality of the resource layer (e.g. monitoring); and (4) *application layer*. In both [13] and [1], as well as in most of the other similar works done by practitioners, the grid architecture is described only at a very high level. The separation between the main parts of the grid is not well defined. Moreover, there is a huge gap between the architectural models, usually given in terms of informal diagrams, and the actual grid implementations which use an engineering-like approach. *A good conceptual reference model for grids is missing.*

This paper tries to fill the gap between high-level architectural diagrams and concrete implementations, by providing a *colored Petri net* (CPN) [14] describing a reference grid architecture. Petri nets [16] are a well established graphical formalism, able to model concurrency, parallelism, communication and synchronization. CPNs extend Petri nets with data, time and hierarchy, and combine their strength with the strength of programming languages. For these reasons, we consider CPNs to be a suitable language for modeling grid architectures. The CPN reference model, being formal, resolves ambiguities and provides semantics. Its graphical nature and hierarchical composition contribute to a better understanding of the whole grid mechanism. Moreover, as CPN models are executable and supported by CPN Tools [11] (a powerful modeling and simulation framework), the model can be used for rapid prototyping, and for all kinds of analysis ranging from model checking to performance analysis.

Literature refers to different types of grids, based on the main applications supported. For example, a *data grid* is used for managing large sets of data distributed on several locations, and a *computational grid* focuses on offering computing power for large and distributed applications. Each type of grid has its particular characteristics making it a non-trivial task to unify them. This paper focuses only on computational grids. However, we also take into account some data aspects, such as input and output data of computational tasks and the duration of data transfer, as these are important aspects for the analysis.

(Computational) grids are used in different domains ranging from biology and physics to weather forecasting and business intelligence. Although the results presented in this paper are highly generic, we focus on *process mining as an*

application domain. The basic idea of *process mining* is to discover, monitor and improve *real* processes (i.e., not assumed processes) by extracting knowledge from event logs [4,3]. It is characterized by an abundance of data (i.e., event logs containing millions of events) and potentially computing intensive analysis routines (e.g., genetic algorithms for process discovery). However, as many of its algorithms can be distributed by partitioning the log or model, process mining is an interesting application domain for grid computing.

At TU/e we are involved in many challenging applications of process mining that could benefit from the grid (a recent example is our analysis of huge logs coming from the "CUSTOMerCARE Remote Services Network" of Philips Medical Systems). We need a good experimental framework that allows us to experiment with different scheduling techniques and grid application designs. To show how our CPN model can be applied in this direction, and that is not only suitable as a descriptive model, we perform a simulation study. Using a small (but typical) process mining application as input we conduct several simple experiments to see how parameters such as the arrival rate, distribution strategies, and data transfer, influence the throughput time of an application and resource utilization. The simulations are done under the realistic hypothesis that the resources are unreliable, i.e., can appear and disappear at any moment in time. For the visualization and analysis of the results we use the link of CPN tools with the SPSS software [2] and ProM framework [5]. Note that in this paper we do not aim to come with a novel middleware design, or invent a new scheduling policy, but rather to illustrate the powerful capabilities of the model and its simulation environment.

The rest of the paper is organized as follows. In the remainder of this section we discuss some related work. In Section 2 we present a grid architecture and its CPN model. The simulation experiments are presented in Section 3. Section 4 concludes the paper.

Related Work. While formal techniques are widely used to describe grid workflows [12,6,8], only a few attempts have been made to specify the semantics of a complete grid. In [15] a semantical model for grid systems is given using Abstract State Machines [7] as the underlying formalism. The model is very high level (a refinement method is only informally proposed) and no analysis is performed. [18] gives a formal semantic specification (in terms of Pi-calculus) for dynamic binding and interactive concurrency of grid services. The focus is on grid service composition.

In order to analyze grid behavior several researchers developed grid simulators. (The most notable examples are SimGrid [10] and GridSim [9].) These simulators are typically Java or C implementations, meant to be used for the analysis of scheduling algorithms. They do not provide a clear reference model as their functionality is hidden in code. This makes it difficult to check the alignment between the real grid and the simulated grid.

In [8] we proposed to model Grid workflows using CPNs, and we also used process mining as a running example. In that paper, however, we fully covered only the application layer of the grid; for the other layers the basic functionality

was modeled, just to close the model and make the analysis possible. We also completely abstracted from data aspects.

2 Modeling a Grid with CPNs

In this section we present a grid architecture and give its semantics using colored Petri nets.[1] The proposed architecture is based on an extensive review of literature. As mentioned in the introduction, CPNs are graphical and have hierarchy, so we use the CPN model itself to explain the architecture.

The main page of the model is given in Figure 1. It shows the high-level view of the grid architecture. As seen in the figure, the grid consists of three layers: (1) the *resource layer*, (2) the *application layer*, and (3) the *middleware*. The application layer is where the users describe the applications to be submitted to the grid. The resource layer is a widely distributed infrastructure, composed of different resources linked via Internet. The main purpose of the resources is to host data and execute jobs. The middleware is in charge of allocating resources to jobs, and of other management issues. The three layers communicate by exchanging messages, modeled as (interface) places in CPNs.

User applications consist of *jobs*, atomic units of work. The application layer sends job descriptions to the middleware, together with the locations of the required input data. It then waits for a message saying whether the job was finished or it was canceled. When it receives a job description the middleware tries to find a resource to execute this job. If a suitable resource is found, it is first claimed and then the job is sent to it. The middleware monitors the status of the job and reacts to state changes. The resource layer sends to the middleware the acknowledgments, and the information on the current state of resources, new data elements, and finished jobs. When instructed by the application layer, the middleware removes the data that is no longer needed from the resources.

We now zoom-in the submodules in Figure 1 and present each layer and its dynamics in more detail.

2.1 Application Layer

The upper level of the grid architecture is composed of user applications. These applications define jobs to be executed on the resources. Since these jobs may causally depend on one another, the application layer needs to specify the "flow of work". Therefore, we use the term *grid workflow* to refer to the processes specified at the application layer. There may be different grid workflows using the same infrastructure, and there may be multiple *instances* of the same grid workflow, referred to as *cases*.

The purpose of a grid architecture is to offer users an infrastructure to execute complex applications and at the same time hide the complexity of the resources. This means that the user should not be concerned with, e.g., resource

[1] We assume that the reader is familiar with the formalism; if otherwise, [14] provides a good introduction.

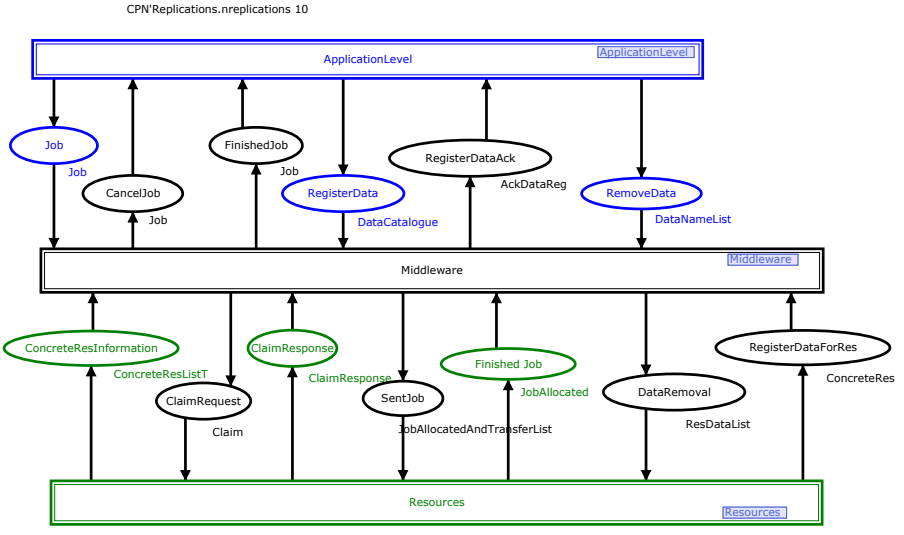


Fig. 1. Grid architecture

discovery or data movement. In order to achieve this goal the application layer provides a grid workflow description language that is independent of resources and their properties. Defined workflows can therefore be (re)used on different grid platforms.

In our case, CPNs are themselves used to model grid workflows. However, they are expected to follow a certain pattern. The user describes each job as a tuple (ID, AP, OD) where ID is the set of input data, AP is the application to be executed, and OD is the set of output data. Every data element is represented by its logical name, leaving the user a possibility to parameterize the workflow at run-time with actual data. All data is case related, i.e. if the user wants to realize multiple experiments on the same data set, the data is replicated and the dependency information is lost. It is assumed that the set ID contains only the existing data, i.e. that at least one resource has this data (e.g. created by previous jobs). It is also assumed that AP is an existing application at the level of resources.

In Figure 2 we present the CPN model of the application layer. In this case the layer consists of only one workflow, but multiple cases can be generated by the *GenCases* module. Every workflow is preceded by the substitution transition *RegisterData*. This transition, as the name implies, registers for every new case the location of the input data required for the first job. The workflow from Figure 2 thus needs a *"Log"* and a *"FilterFile"*. When all the jobs in a case are executed, the application layer sends a message to the middleware instructing that all the data of the case is deleted (transition *Garbage Removal*).

In Figure 3, an example of a workflow is presented, describing a simple, but very typical, process mining experiment. The event log (*"Log"*) is first filtered using a filter described in *"Filter File"*. Then, the obtained *"Filtered Log"* is

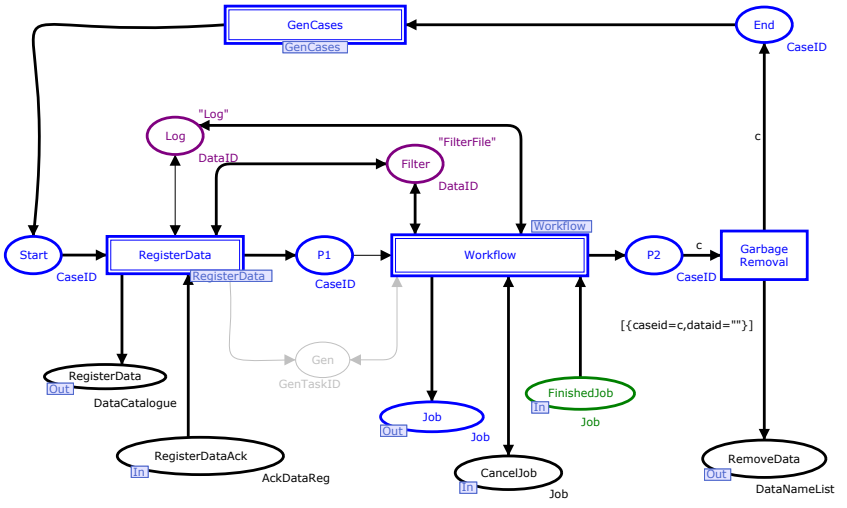


Fig. 2. Application Layer

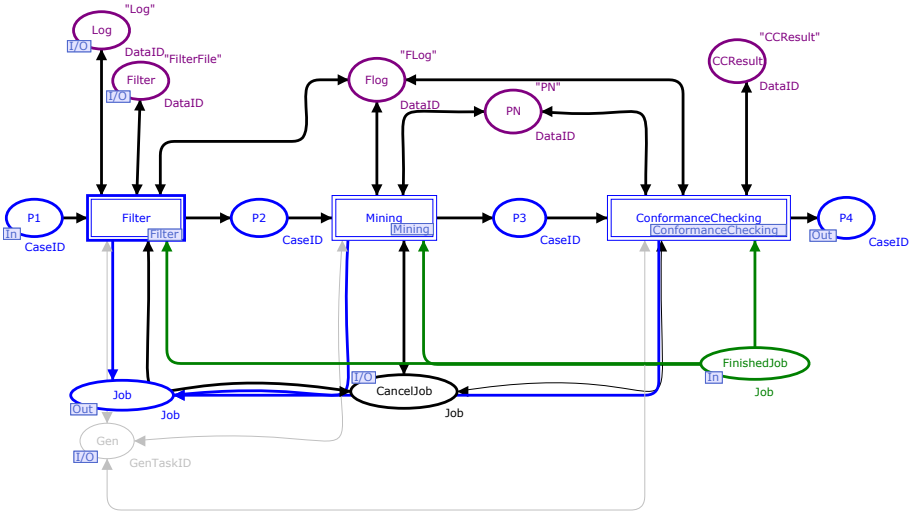


Fig. 3. Workflow example

mined and the result of the mining algorithm (*"PN"*) is assessed by using the conformance checker (to see, e.g. how many traces from the log can be reproduced by the mined model).

All jobs follow the pattern from Figure 4. Each logical data name (from ID and OD) is modeled as a distinct place. In this way we can easily observe the data dependencies between jobs. These dependencies can be used in an optimization algorithm to decide whether some data is no longer needed, or when some data

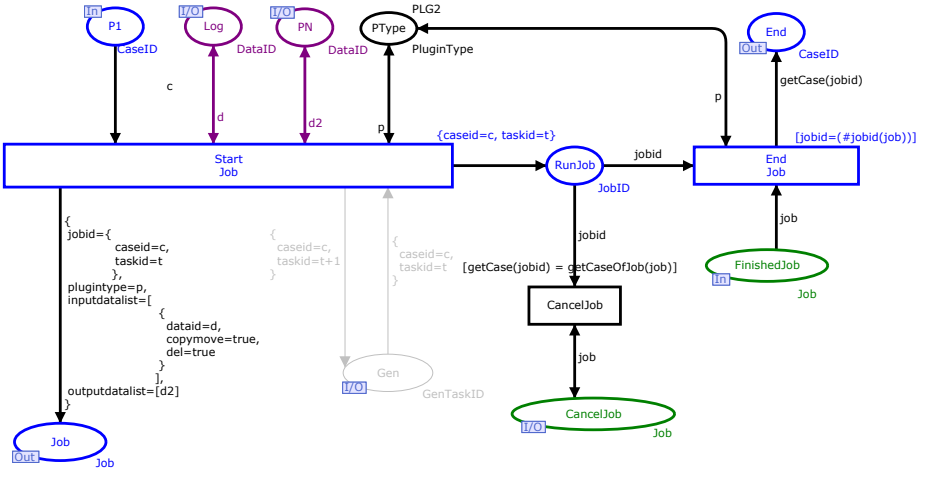


Fig. 4. Job example

is more suitable to be replicated. The *PluginType* place contains the information on which application to execute at the resource level (the parameter AP from the above). In our case this place contains the name of a process mining algorithm. Every job starts by receiving a unique id and sending its description to the middleware. It ends with either a proper termination (the job was executed and required output data was created), or a cancelation (the middleware can not find a resource to execute the job; in our model that only happens when some input data does no longer exist on any grid resource). The application layer cancels the whole case if at least one of the jobs gets canceled.

2.2 Middleware

The link between user applications and resources is made via the middleware layer (Figure 5). This layer contains the intelligence needed to *discover*, *allocate*, and *monitor* resources for jobs. We consider just one centralized middleware, but our model can be easily extended to a distributed middleware. We also restrict ourselves to a middleware working according to a "just-in-time" strategy, i.e., the search for an available resource is done only at the moment a job becomes available. If there are multiple suitable resources, an allocation policy is applied. Look ahead strategies and advanced planning techniques are not considered in this paper.

The place *GlobalResInformation* models an information database containing the current state of resources. The middleware uses this information to match jobs with resources, and to monitor the behavior of resources. The database is updated based on the information received from the resource layer and the realized matches.

Data Catalog is a database containing information about the location of data elements and the amount of storage area that they occupy. This information is also used

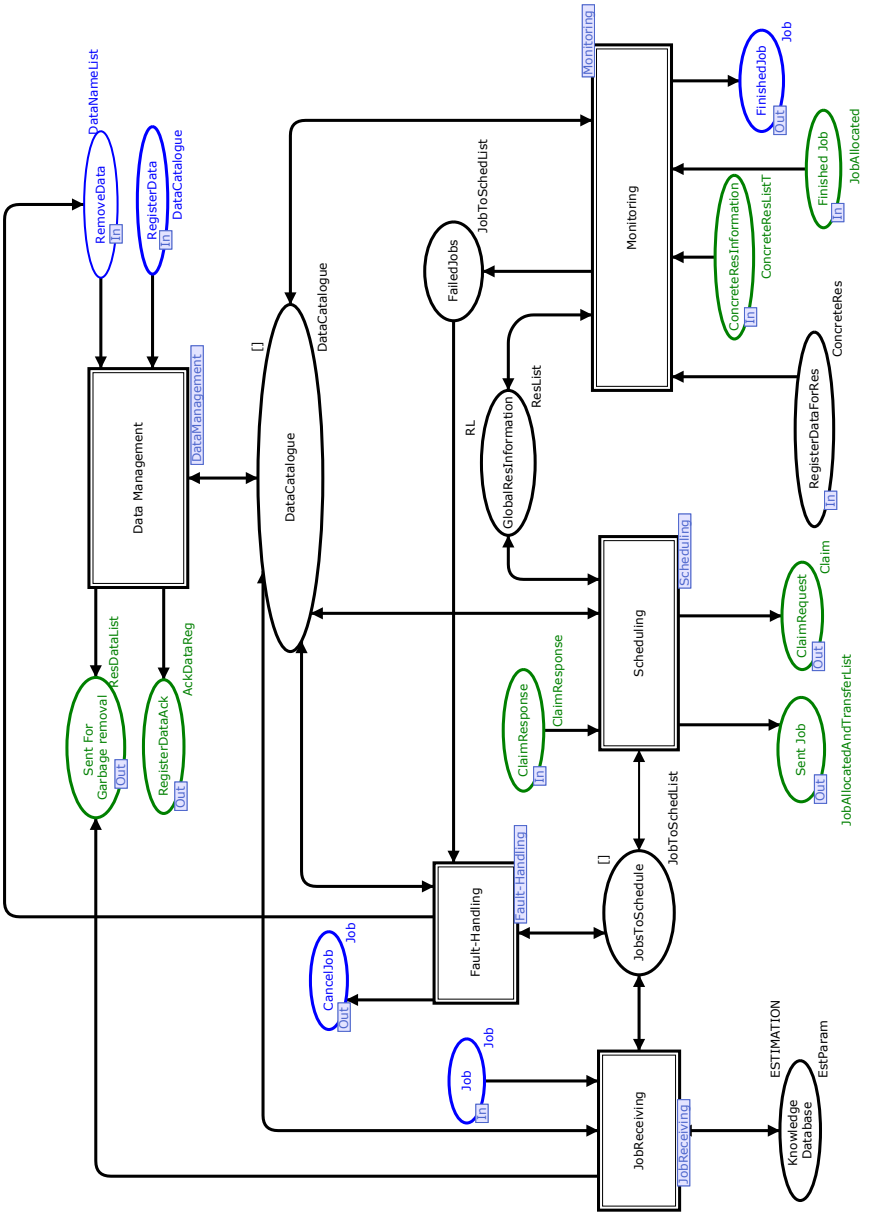


Fig. 5. Middleware layer

in the scheduling process. The transition *DataManagement* models the registration of data for new cases and the removal of records from the catalogue. A message containing a list of "garbage data" can be sent to the resources at any time.

When a job is received (*JobReceiving* module), the middleware first extends its description with an estimate of how much storage area is needed. A user-provided knowledge database is used for this task. Next, the job is added to the jobs pool list, ordered based on the arrival time. If multiple jobs arrive at the same time, the order is non-deterministic. The scheduling process now starts (Figure 6), according to the chosen policy.

Scheduling is done in two steps. First, a match between a resource and a job is found. The matching phase succeeds is there is a resource with a free CPU and enough free space to store the input and the output data. Second, the found resource is claimed. This step is necessary because the matching process is based on a (possibly outdated) local copy of the state of the resources. The middleware sends a claim request to the allocated resource in order to check that its resource image is still correct. If the claim is successful, the middleware sends the job description to the resource, extended with the list of data that need to be obtained from other resources (using the so-called transfer list). If the claim fails, the middleware puts the job back into the pool.

After the job was sent to the resource, the middleware monitors the resource state. Basically, it listens on the signal received from the resource layer. Each time a message is received, the middleware compares the received state with its local copy. Since these messages can be outdated (e.g., when a resource sends a message just before a job arrives), the middleware reacts using a timeout mechanism. A resource is considered unavailable if no information is received from it for a given period of time, and a job is considered canceled if no message related to the job is received for several consecutive updates.

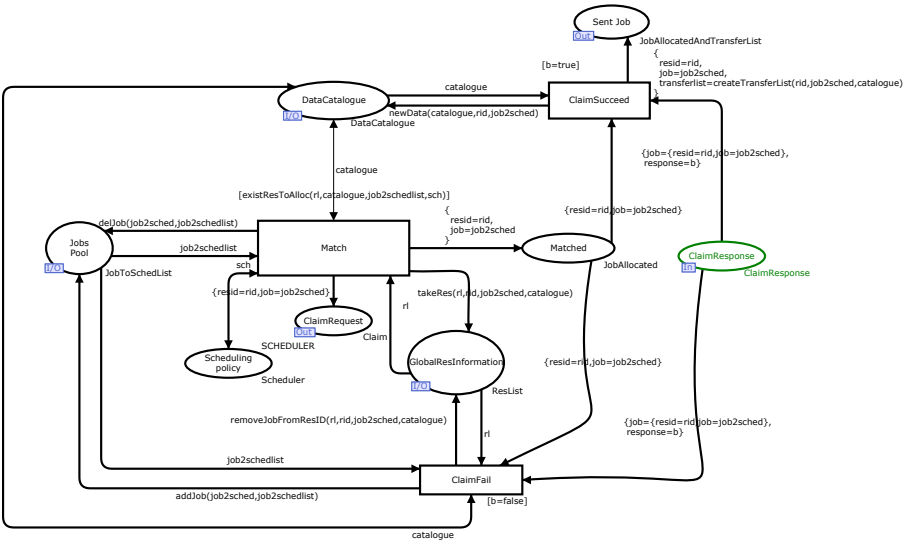


Fig. 6. Scheduling page

When the middleware receives the message that a job is finished, it updates the global resource information database and forwards this message to the application layer.

Jobs can fail at the resource layer. Therefore, a fault handling mechanism is defined (transition *Fault-Handling*). When a job fails, the middleware tries to re-allocate it. However, if the necessary input data is no longer available at the resource level, the middleware is unable to execute the job and it sends a message to the application layer that the job is canceled.

2.3 Resource Layer

Every resource is described in terms of the available computing power (expressed in number of CPUs), the amount of storage area available for hosting data, the list of supporting applications, and the set of running and allocated jobs. The resources are unaware of job issuers and of job dependencies. Every job is executed on just one resource. However, resources can work on multiple jobs at the same time. Figure 7 presents the conceptual representation of the functionalities of the resource layer in terms of CPNs.

The set of resources is assumed to be fixed, but resources are considered unreliable. They can appear/dissapear at any moment in time, except when transferring data. Transition *Resource Dynamics*, governed by a stochastic clock, simulates the possibility that a resource becomes unavailable. When this happens all the data is lost and all running jobs are aborted on this resource.

After a successful match by the middleware, the transition *Claim* is used to represents a guarantee that the allocated resource can execute the job. Recall that this phase is necessary because the allocation at the middleware level is based on a possibly outdated information of the state of the resources. If the claiming succeeds, one CPU and the estimated necessary storage area are reserved at the resource. The resource is now ready to perform the job, and is waiting for the full job description to arrive. The job description also contains the locations of the input data and the information on which application to execute. The substitution transition *Transfer* models the gathering of necessary input data from other resources. If the input data is no longer present on a source node, the job is aborted. If the transfer starts, we assume that it ends successfully. Note that the reserved CPU remains unoccupied during the transfer. When all the input data is present on the allocated resource, the job starts executing.

The resources can always offer their capabilities and, so, the resource layer constantly updates the middleware on the current state of the resources. There are two types of updates sent: (1) recognition of new data (transferred data, or data generated by job execution) and (2) signals announcing to the middleware that a resource is still available. While the former is sent on every change in the resource status, the latter info is periodical.

Remove Data transition models the fact that any data can be deleted from a resource at the request of the middleware. These requests can arrive and be fulfilled at any moment in time.

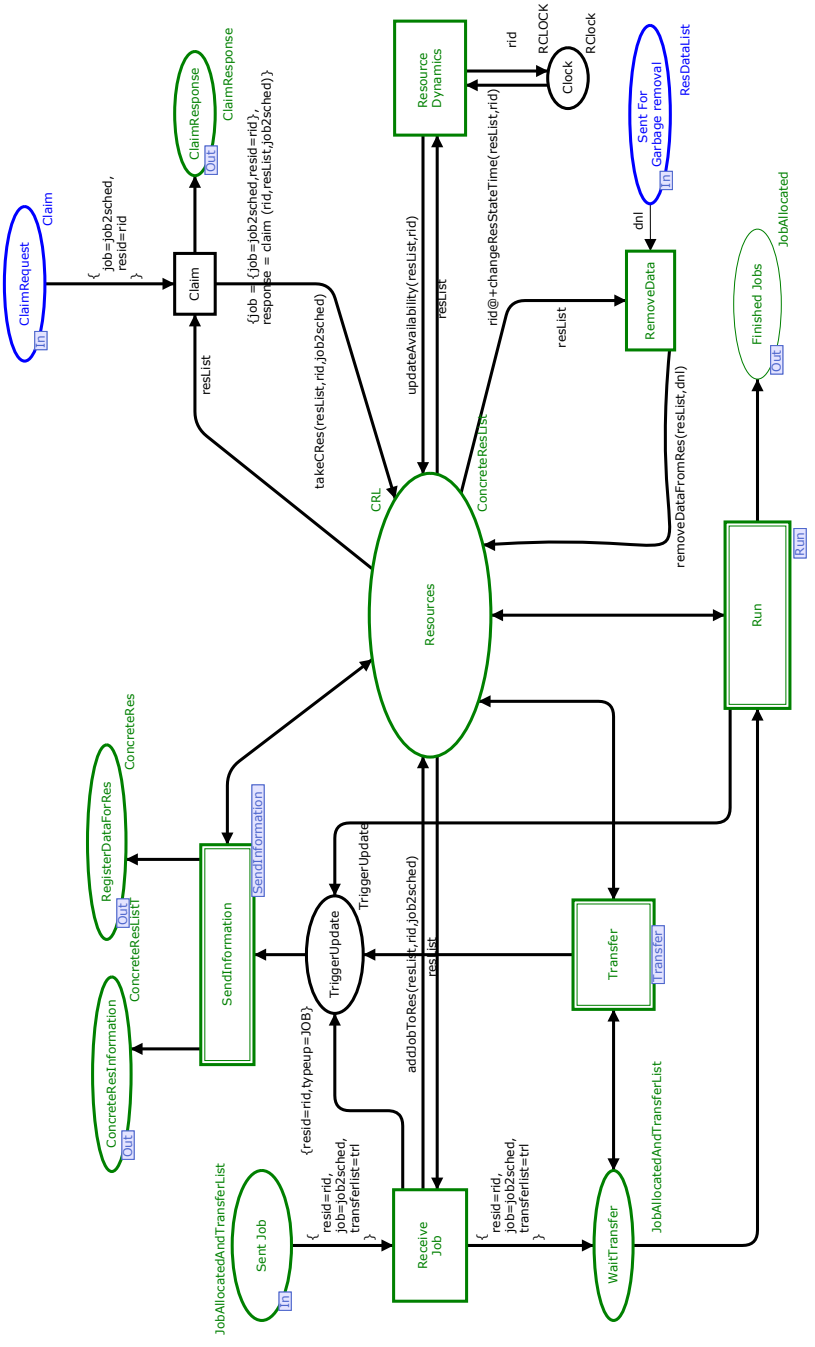


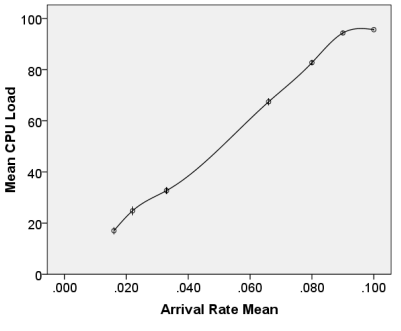
Fig. 7. Resource layer

The reference model presented in this section offers a clear view and a good understanding of our grid architecture. The next section shows how we can use this model to also analyze the behavior of the grid.

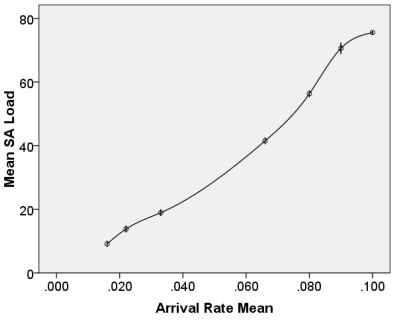
3 Simulation Analysis of the CPN Model

In order to measure performance of the grid we perform an extensive simulation study on the reference model. The simulations are done using the capabilities of CPN Tools. We use the SPSS software [2] and ProM framework [5] to visualize and analyze the results. SPSS is a known package for statistical analysis; ProM is a powerful, extendable and open-source, framework supporting model discovery, analysis, and conversion. The purpose of this section is mostly to show the reader what kind of analysis can be done using the model, and to illustrate the powerful capabilities of the environment.

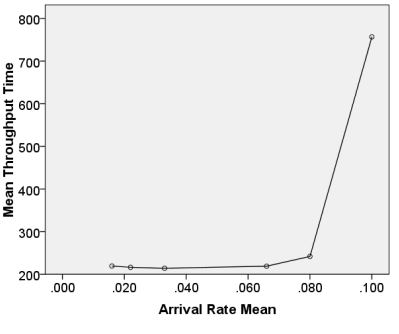
Our main goal is to see how does the grid behave under the heavy load of process mining applications. The measures of interest are the average *throughput time* of application (user view), and the average *utilization* of resources (system view). We first thoroughly explore the behavior of the model based on the simulation data. Then, we try to improve our scheduling policy by changing the data transfer strategy.



(a) CPU Load



(b) Storage Area Occupancy



(c) Throughput Time

Fig. 8. Simulation Results

The testbed for our experiment is as follows. We consider a resource pool containing 6 identical resources, each having 3 CPUs and a storage area of 1000GB. The resources are unreliable, and can appear/disappear at any moment. Their dynamics is governed by a uniform distribution. We assume that the resources are used exclusively for our process mining applications. Every resource can perform the three process mining operations, i.e. *Filtering*, *Mining* and *Conformance Check*. All user applications follow the workflow structure from Figure 3. The individual cases arrive according to an exponential distribution, and have uniformly distributed input file (i.e. of the log and the filter file) sizes . We take the scheduling policy to be *first-ready-first-served*, but the scheduling algorithm gives priority to the more advanced cases, i.e., *Conformance Check* jobs have higher priority than *Filtering* jobs. The motivation for this comes from the fact that garbage removal takes place only at case completion.

The performance measures in question are examined for job arrival rates of 2,4,6,8, and 10, jobs per 100 time units. We perform 10 independent simulations for each of the examined configurations, and we calculate 95% confidence intervals. Each simulation run is limited to 2000 jobs.

In our first calculation we assume that data required for a job is always copied to the allocated resource, and never moved (i.e., it stays on the source resource). This strategy, on the one side, overloads the grid with a lot of replicated data and, therefore, reduces performance. On the other side, however, it gives the middleware more options when allocating a job, thus improving the performance.

Figure 8 shows the evolution of the performance parameters when the arrival rate is varied. Figures 8(a) and 8(b) show the evolution of the resource utilization, in terms of the number of CPUs used (called CPU load when in percentage) and the amount of storage area occupied; Figure 8(c) shows the evolution of

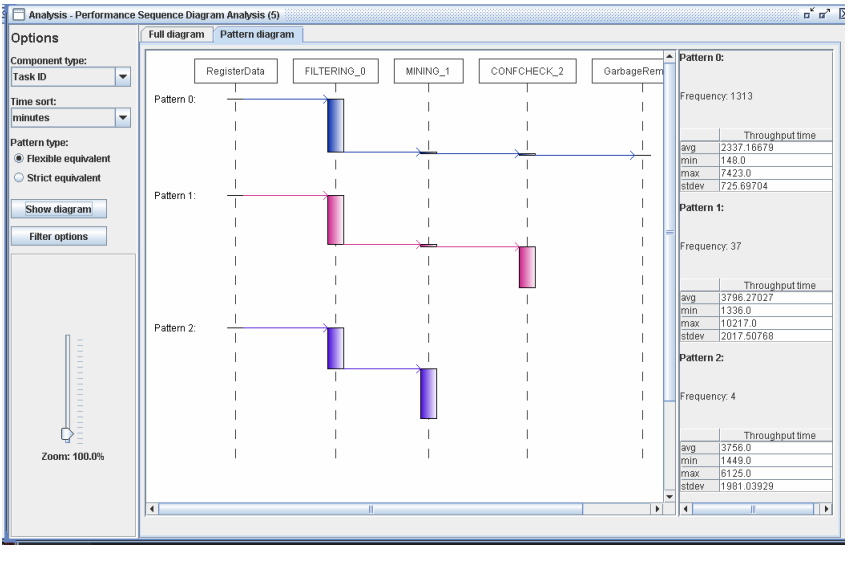


Fig. 9. *Performance Sequence Diagram* showing the execution patterns

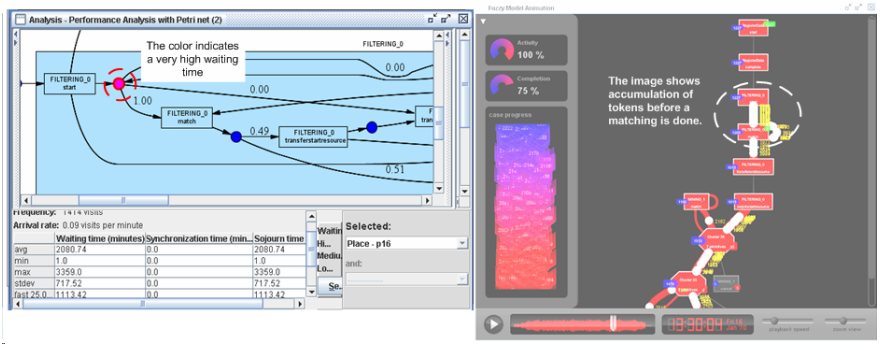


Fig. 10. Bottlenecks found with *Performance Analysis with Petri Nets* plugin and *Fuzzy Miner*

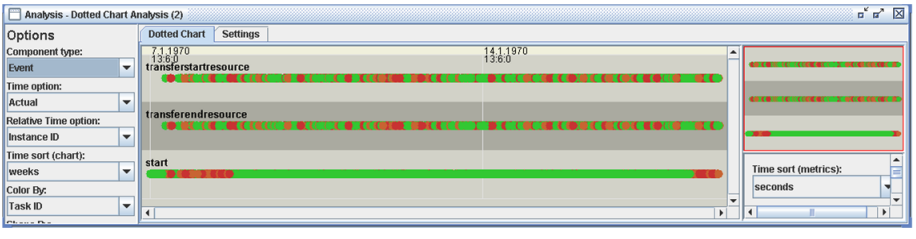
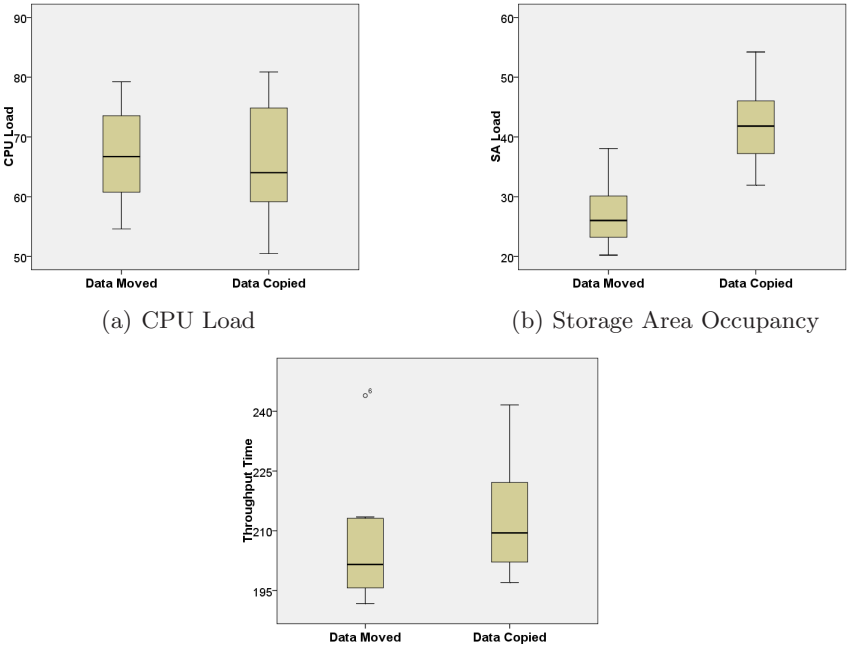


Fig. 11. *Dot Plot* plugin showing the transfer event frequency

the throughput time. We observe that when the CPU load is less then 80%, the throughput time is around 200 time units for all the arrival rates. When the arrival rate is around 8/100, the resource load stabilizes to around 100%, but the throughput time starts to increase swiftly.

To find bottlenecks we do a more detailed inspection using ProM. We first apply the *Performance Sequence Diagram* plugin, which gives us the result shown in Figure 9. The chart represent the individual execution patterns for the case of the highest arrival rate. We observe that the execution time for filtering (including the queueing time) is higher than for the other jobs. As the resource occupation is very high, the newly arrived cases wait long to be scheduled. The patterns 2 and 3 are the *cancelation* executions. The execution time value for canceled jobs is higher than for those with a normal execution (Pattern 1). This is because the middleware cancels jobs based on a time out mechanism. Similar conclusions can be made by using the *Performance Analysis with Petri Nets* plugin, and the *Fuzzy Miner* plugin, as seen in Figure 10. Using the *Dot Plot* plugin (Figure 11) we observe that when the arrival rate is 1/10, the frequency of data transfer is significantly higher than for the lower arrival rates. As this arrival rate is very high, after a job is finished the next job of the same case is unlikely to be scheduled on the same resource.

In our second experiment we change the data transfer strategy, and no longer replicate the data but move it. Figure 12 shows the confidence intervals for the



(a) CPU Load

(b) Storage Area Occupancy

(c) Throughput Time

Fig. 12. Comparison between the moving data strategy and replication strategy

two strategies, when the arrival rate is 1/10. With the new strategy storage area occupation is decreased by half, and there is a slight improvement in the throughput time.

4 Conclusions

In this paper, we presented a reference model for grid architectures in terms of colored Petri nets, motivated by the absence of a good conceptual definition for the grid. Our model is formal and offers a good understanding of the main parts of the grid, their behavior and their interactions. To show that the model is not only suitable for definition purposes, we conducted a simulation experiment. Under the assumption that the grid is used for process mining applications, we compared the performance of two scheduling strategies, different in the way they handle data transfer.

The grid model is the starting point in the developing of both an experimental simulation framework and a real grid architecture to support process mining experiments.

References

1. Grid architecture,
`http://gridcafe.web.cern.ch/gridcafe/gridatwork/architecture.html`

2. SPSS software, http://www.spss.com/
3. Aalst, W., Reijers, H., Weijters, A., van Dongen, B., Medeiros, A., Song, M., Verbeek, H.: Business Process Mining: An Industrial Application. Information Systems 32(5), 713–732 (2007)
4. Aalst, W., Weijters, A., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
5. van der Aalst, W., van Dongen, B., Günther, C.W., Mans, R., de Medeiros, A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.E., Weijters, A.: ProM 4.0: Comprehensive support for real process analysis. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 484–494. Springer, Heidelberg (2007)
6. Alt, M., Gorlatch, S., Hoheisel, A., Pohl, H.-W.: A grid workflow language using high-level Petri nets. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 715–722. Springer, Heidelberg (2006)
7. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
8. Bratosin, C., van der Aalst, W.M.P., Sidorova, N.: Modeling grid workflows with colored Petri nets. In: Proceedings of the Eighth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2007). DAIMI, vol. 584, pp. 67–86 (October 2007)
9. Buyya, R., Murshed, M.M.: 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)
10. Casanova, H.: Simgrid: A toolkit for the simulation of application scheduling. In: CCGRID 2001: Proceedings of the 1st International Symposium on Cluster Computing and the Grid, Washington, DC, USA, p. 430. IEEE Computer Society, Los Alamitos (2001)
11. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page, http://wiki.daimi.au.dk/cpntools/
12. Feng, Z., Yin, J., He, Z., Liu, X., Dong, J.: A novel architecture for realizing grid workflow using pi-calculus technology. In: Zhou, X., Li, J., Shen, H.T., Kitsuregawa, M., Zhang, Y. (eds.) APWeb 2006. LNCS, vol. 3841, pp. 800–805. Springer, Heidelberg (2006)
13. Foster, I.: The anatomy of the grid: Enabling scalable virtual organizations. In: Proceedings of First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001, pp. 6–7 (2001)
14. Jensen, K.: Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical. Springer, Heidelberg (1992)
15. Nemeth, Z., Sunderam, V.: A formal framework for defining grid systems. In: Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2002, Berlin (2002)
16. Reisig, W.: System design using Petri nets. In: Requirements Engineering, pp. 29–41 (1983)
17. Stockinger, H.: Defining the grid: a snapshot on the current view. The Journal of Supercomputing 42(1), 3–17 (2007)
18. Zhou, J., Zeng, G.: Describing and reasoning on the composition of grid services using pi-calculus. In: CIT 2006: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology (CIT 2006), Washington, DC, USA, pp. 48–54. IEEE Computer Society, Los Alamitos (2006)