Fault-Tolerant Multiuser Computational Grids Based on Tuple Spaces

Fábio Favarim † Joni da Silva Fraga† Lau Cheuk Lung§ Miguel Correia‡
† Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil
‡ LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
§ Prog. de Pós-Graduação em Informática Aplicada, Pontifícia Universidade Católica do Paraná – Brazil
fabio@das.ufsc.br *fraga@das.ufsc.br* *lau@ppgia.pucpr.br* *mpc@di.fc.ul.pt*

Abstract

This paper proposes GridTS, a grid infrastructure in which the resources select the tasks they execute, instead of a scheduler finding resources for the tasks. This solution allows scheduling decisions to be made with up-to-date information about the resources. GridTS provides fault-tolerant scheduling by combining a set of fault tolerance techniques to tolerate crash faults in any components of the system. The communication is supported by a tuple space.

1 Introduction

Heterogeneous resources connected to the Internet are being increasingly used for executing resource-intensive applications, something usually called Grid Computing [1]. The idea is to obtain large amounts of processing power by harnessing idle resources on the Internet. These resources can join and leave the grid at any time. This dynamic nature allows the grids to contain many machines that would not be available otherwise, but on the other hand, it also requires the infrastructure to manage the uncertainty about the availability of the resources and their failure.

This paper is about **scheduling** and **fault tolerance** in multiuser grids that execute **bag-of-tasks** applications [2]. This is only one among several types of applications that can be executed in grids, but there are many important applications that fit in this category: data mining, simulations, massive searches, image processing, etc. These applications are composed by sets of decoupled tasks that can be executed independently, without any kind of synchronization or communication among them. This independence makes these applications specially suited for dynamic grids, since the failure of resources that are executing tasks can easily be handled by rescheduling the interrupted tasks in other resources.

An important goal of any grid infrastructure is to use the resources efficiently, maximizing the resource utilization while trying to minimize the total time to execute a **job**. Each job consists of a set of **tasks**, and each task has to be executed by one of the grid resources for a certain time. A schedule is an assignment of the tasks of a job to a set of resources.

Schedulers can be classified in two classes: **knowledge-based schedulers**, which rely on accurate information about resources' attributes (CPU speed and load, memory) and availability to do the scheduling [3, 4, 5]; and **knowledge-free schedulers**, which do not rely on that information [4, 6]. Starting with the latter, a knowledge-free scheduler, like the Workqueue, works like a queue, picking a task and assigning it to a resource following some order of the resources. When a resource finishes executing a task, it returns the result and the scheduler gives it the next one. A problem with this scheme is that when a large task (e.g., in terms of CPU time and memory needed) is given to a slow resource, the execution of this task may delay the whole job. Knowledge-based schedulers avoid this problem by taking into account information about the resources. The main differences in these scheduling algorithms is in how they compute task priorities. Dynamic FPLTF (a variation of FPLTF for grid computing [3]) gives higher priority to the largest task, assigning larger tasks to faster nodes [4]. MFTF gives more priority to the task most suited for each resource [5]. The algorithm uses the expected task execution time to know how suited a task is to a resource. The information used by knowledge-based schedulers is usually provided by an **information service** that is responsible for gathering information about all resources that compose the grid. Gathering that information is like taking a snapshot of the grid, i.e., getting the global grid state in a certain instant. This operation is reasonably costly in a large grid, and the snapshot tends to become outdated in a short time when the grid is comprised by a large number of non-dedicated, heterogeneous, widely-dispersed resources.

This paper proposes **GridTS**, an infrastructure that provides a knowledge-based scheduling solution in which the resources select the tasks they execute, instead of the scheduler finding resources for the tasks. This solution allows scheduling decisions to be made with up-to-date information, since, naturally, each resource has always up-to-date information about itself. Therefore our solution overcomes the problems of getting up-to-date information about resources faced by knowledge-based schedulers. Nevertheless, a knowledge-based scheduling heuristic that uses only local information can still be plugged in GridTS.

GridTS is based on the **generative coordination** model, in which processes (brokers, resources) interact through a shared memory object called **tuple space** [7]. This coordination model supports communication that is decoupled both in time and space, i.e., in which processes do not need to be active at the same time and do not need to know each others locations or addresses. This makes it particularly suited for highly dynamic systems like a grid.

In GridTS, a user gives its job to a broker that breaks it in tasks and inserts them in the tuple space. The resources are permanently in a loop: get a task from the tuple space, execute the task, put results in the tuple space, get another task, execute task...

In large-scale grids, the probability of failures happening is high. Many of the current grids have single points of failure, i.e., not all their components are fault-tolerant. GridTS is fault-tolerant and has no single points of failure. Fault tolerance is enforced using a combination of mechanisms. Transactions are used to guarantee that the failure of a resource or a broker does not cause the loss of a task or leaves the tuple space in an inconsistent state. Checkpointing is used to limit the work lost when a resource fails during the execution of a task, allowing another resource to continue where the first left. Finally, replication is used to enforce the fault-tolerance and availability of the tuple space. We consider only **crash faults**, which in this context can be accidental (some machine really crashes) or forced by a resource owner that wants to remove his/her machine(s) from the grid. We do not consider the possibility of the resources returning results that do not correspond to the execution of the tasks they are supposed to execute, on the contrary to [8].

This work has two main contributions. Firstly, it presents an architecture for a computational grid that allows resources to find tasks suited for their attributes, even if those attributes change with time. This eliminates the complexity of gathering information about the whole grid. Secondly, the infrastructure provides fault-tolerant scheduling by combining a set of fault tolerance techniques to tolerate crash faults in any component of the system.

2 Tuple Spaces and Fault Tolerance

GridTS is based on a tuple space, a notion first introduced in the Linda programming language [7]. A tuple space can be viewed as a shared memory object that allows distributed processes to interact by inserting tuples in the space. A tuple is an ordered sequence of typed fields. A tuple t in which all fields have values defined is called an **entry**, otherwise it is called a **template**, and it is denoted, e.g., $\bar{t}$.

A tuple space provides three basic operations. The $out(t)$ operation is used to insert a tuple t in the space. The $rd(\bar{t})$ operation reads a tuple t that matches the template $\bar{t}$ from the tuple space (non-destructive read). An entry t and a template $\bar{t}$ match essentially if: they have the same number of fields; the type of the corresponding fields are the same in both; the values in defined fields of $\bar{t}$ are identical to the values in the corresponding fields of t. The $in(\bar{t})$ operation reads *and* removes a tuple from the tuple space (destructive read). Both the in and rd operations are blocking, but they have also non-blocking versions: $inp()$ and $rdp()$. All these read operations are non-deterministic, because if there is more than one matching tuple available, one of them is chosen arbitrarily. There is one more version of $rd(\bar{t})$ that returns *all* the tuples that match $\bar{t}$: $copy\_collect(\bar{t})$ [9]. Note that an important characteristic of this coordination model is the associative nature of the communication: the look up for tuples on the tuple space is based on their content, rather than accessed through an address or identifier.

Fault tolerance in the generative coordination model can be considered at two levels:

- *fault tolerance of the tuple space*, i.e., the problem of guaranteeing that the space does not fail if there are faults in the tuple space itself; and
- *application-level fault tolerance*, i.e., to ensure that the applications satisfy certain dependability properties even if some of the applications' processes fail.

In GridTS, the first issue is handled using replication, i.e., by running the tuple space in several servers and ensuring that the tuple space as a whole tolerates the failure of some of the servers [10, 11]. In this paper we consider that the tuple space is indeed implemented by a set of servers and is fault-tolerant but we do not delve into the details of how it is done since the problem is well understood and is solved.

The second issue is handled by application-level fault tolerance mechanisms provided by the tuple space, usually **transactions** [12, 11]. This mechanism guarantees essentially that if a process tries to execute a set of operations in the tuple space, either all the operations are executed or none of them is. Several currently available tuple spaces, like JavaSpaces and TSpaces, provide this mechanism.

3 GridTS

3.1 The Infrastructure

An overview of the GridTS infrastructure is shown in Figure 1. The basic functioning is based on the **master-workers** pattern [10, 11]. This pattern has two kinds of entities: one master and several workers. The master gives tasks to the workers that execute them and return the results to the master. In GridTS there is not one but several masters – called **brokers** – that get **jobs** from the **users**, divide them in **tasks** and give these tasks to the workers, which are the grid's **resources**. Brokers are usually specific to one class of applications, i.e., they only know how to divide in tasks jobs of this class.

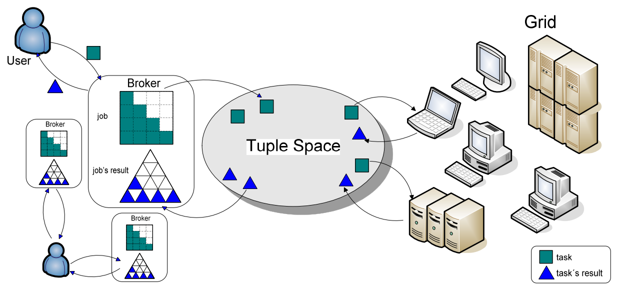

Figure 1. The GridTS infrastructure

GridTS use a tuple space to support the scheduling. Briefly, the idea is the following:

- The user submits a job to a broker that decomposes the job in several small tasks. The broker insert tuples describing these tasks in the tuple space (**task tuples**).
- The resources retrieve from the space tuples that describe tasks they are able to execute, and execute them. After each execution, the result is placed in the tuple space.
- When all job's tasks finish executing, the broker assembles all task results and gives them to the user that submitted the job.

Each task represents one unit of work that may be performed in parallel with other tasks. The tuple that describes a task contains all the necessary information for its execution: the identification of the task, the requirements for its execution (e.g. processor load, processor speed, available memory, operating system), the code to be executed (or the location from where to download it), and the parameters – input data– to the execution of the task (or their location). The users do not need to know which resources will execute the tasks, their location or when these resources will be available.

3.2 Fairness and Fault Tolerance

The architecture of GridTS has the immediate benefit of not requiring an information service to give information about the resources (their availability, usage and other parameters), on the contrary of most grid platforms. GridTS enforces a natural form of **load balancing** since the resources pick tasks adequate to their conditions and get a new one whenever the previous ended. However, there are also some challenges. The first is a problem of fairness since multiple users/brokers can put tasks concurrently in the tuple space. The second is related to fault tolerance: GridTS has to tolerate failures in the brokers and, more importantly, to deal efficiently with failures of the resources.

To guarantee a **fair scheduling**, the resources have to pick tasks from the tuple space using an appropriate mechanism. A solution that ensures fairness is to use a *ticket*. When a broker wants to insert a tuple in the space, it picks a tuple that represents the ticket (**ticket tuple**), inserts the job with the current ticket number, increases the ticket number and inserts it back in the space. A job is represented in the tuple space by a **job tuple** and a set of task tuples. When a resource wants to pick a task, the criteria should be to select the job with lowest ticket and then a task from that job, to ensure fairness. However, other criteria are possible, like using a *network of favors*, where the users who donate more resources will have greater priority when they need to make use of the grid [13]. This solution does not ensure fairness in the sense we were considering but in a sense more related to Economy.

Transactions are used by both brokers and resources. Broker use transactions to ensure that: (1) the job's tasks are insert atomically in the space, i.e., either they are all inserted or none is (in case the broker fails during the insertion); (2) the ticket is not lost if the broker removes it and crashes before inserting it back incremented in the space; (3) to get the results of the tasks atomically from the space. On the resources-side, transactions are used mainly to guarantee that when a resource fails during the execution of a task, the task tuple is returned to the space to be eventually executed by another resource (or the same if it recovers).

Tasks usually take a long time to execute, e.g., hours or even days, so it is not convenient to restart from scratch the execution of a task whenever the resource that is executing it fails. To minimize this problem, GridTS uses a backward error recovery mechanism that consists in periodically saving the state of the task execution – a **checkpoint** – in the tuple space [14]. If the resource fails, then another resource continues the execution of the task from that checkpoint, thus limiting the work lost when a resource fails during the execution of a task.

The execution of checkpoint requires the use of **nested transactions**. A task is executed in the context of a parent transaction, which returns the task tuple to the space in case of failure. The execution of the task between two checkpoints is also done in the context of a sub-transaction, to return the **transaction tuple** back to the tuple space in case the resource fails. This scheme is based on **nested top-level transactions** [15], because if the parent transaction aborts, the effects of the sub-transactions – the checkpoint tuples inserted in the space – persist.

3.3 Evaluation

We are currently evaluating the performance of GridTS using the GridSim simulator [16]. We are comparing GridTS with a grid with a knowledge-free scheduler and two grids with knowledge-based schedulers (based on a centralized information service). The former simply blindly scatters the tasks by the resources. The latter are based respectively on the MFTF and the Dynamic FPLTF heuristics [5, 4].

A first set of simulations has shown very similar performances for all the four scheduling schemes. This somewhat unexpected result comes from the assumptions made in the simulations: all resources had the same attributes (CPU speed, memory, load) and there were no failures. These conditions do not permit the simulations to show the advantages of using knowledge about the resources, which should appear when comparing the knowledge-based schedulers with the knowledge-free scheduler. They also do not permit to see the benefits of always having fresh information about the resources (in GridTS) over having information that may be somewhat old or hard to collect (in the knowledge-based schedulers).

We are currently extending the simulations to observe these advantages. Preliminary results have shown already that the benefits of GridTS over knowledge-based schedulers depend heavily on the timeout used to detect the failure of a resource, the probability of a resource failing, and the heterogeneity and dynamicity of the resources conditions. We also envisage comparing GridTS with the scheme for dynamically allocating and reallocating resources proposed in [17].

4 Conclusion

This paper presents GridTS, a decentralized and fault-tolerant grid infrastructure. Instead of using a centralized scheduler, the resources are in charge of picking the tasks they will execute. The communication is based on the generative coordination model, i.e., on a shared memory object called a tuple space. The solution combines different fault tolerance techniques, like transactions, checkpointing and replication.

Future work will focus on designing a secure version of GridTS. We envisage extending GridTS with security mechanisms like access control, to guarantee, e.g., that the results of a job can be read only by the corresponding broker. Moreover, we expect to use intrusion tolerance mechanisms to guarantee the security of the system even if some of its components are compromised. Finally, the scalability of the infrastructure will be improved by designing a scalable tuple space.

5 Acknowledgments

The work described in this paper was partly supported by the CNPq (Brazilian National Research Council) through process 140848/2003-7 and by the CAPES through process 3622/05-3.

References

[1] Ian Foster and Carl Kesselmann, Eds., *The GRID: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1999.

[2] J. A. Smith and S. K. Shrivastava, “A system for fault-tolerant execution of data and compute intensive programs over a network of workstations,” in *Proceedings of the 2nd International Euro-par Conference (EURO-PAR’96)*, 1996, number 1123 in LNCS, pp. 487–495.

[3] Daniel A. Menascé, Debanjan Saha, Stella C. da Silva Porto, Virgilio A. F. Almeida, and Satish K. Tripathi, “Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures,” *Journal of Parallel and Distributed Computing*, vol. 28, no. 1, pp. 1–18, 1995.

[4] Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro, “Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids,” in *Proceedings of the 9th International Euro-par Conference (EURO-PAR’03)*, 2003.

[5] Sheng-De Wang, I-Tar Hsu, and Zheng Yi Huang, “Dynamic scheduling methods for computational grid environments,” in *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS’05)*, 2005, pp. 22–28.

[6] Noriyuki Fujimoto and Kenichi Hagihara, “Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid,” *Annual International Conference on Parallel Processing (ICPP-03)*, vol. 00, pp. 391–398, 2003.

[7] David Gelernter, "Generative communication in Linda," *ACM Transactions on Programing Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[8] Paul Townend, Paul Groth, Nik Looker, and Jie Xu, "Ft-grid: A fault-tolerance system for e-Science," in *Proceedings of the 4th UK e-Science All Hands Meeting (AHM05)*, 2005.

[9] Antony I. T. Rowstron and Alan Wood, "Solving the Linda multiple rd problem using the copy-collect primitive," *Science of Computer Programming*, vol. 31, no. 2-3, pp. 335–358, 1998.

[10] Andrew Xu and Barbara Liskov, "A design for a fault-tolerant, distributed implementation of Linda," in *Proceedings of the 19th Symposium on Fault-Tolerant Computing (FTCS'89)*, 1989, pp. 199–206.

[11] David E. Bakken and Richard D. Schlichting, "Supporting fault-tolerant parallel programming in Linda," *IEEE Transactions on Parallel and Distributed Systems*, vol. 06, no. 3, pp. 287–302, 1995.

[12] Karpjoo Jeong and Dennis Shasha, "Plinda 2.0: A transactional/checkpointing approach to fault tolerant Linda," in *Proceedings of the 13th Symposium on Reliable Distributed Systems*, 1994, pp. 96–105.

[13] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg, "OurGrid: An approach to easily assemble grid with equitable resource sharing.," in *Proceedings of 9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

[14] Peter A. Lee and Thomas Anderson, *Fault Tolerance: Principles and Practice*, Dependable Computing and Fault-Tolerant System. Springer-Verlag, 2nd edition, 1990.

[15] Barbara Liskov and Robert Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Transactions on Programing Languages and Systems*, vol. 5, no. 3, pp. 381–404, 1983.

[16] Rajkumar Buyya and Manzur Murshed, "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, vol. 14, no. 13–15, pp. 1175–1220, 2002.

[17] Charles'Kubicek, Mike Fisher, Paul McKee, and R. Smith, "Dynamic allocation of servers to jobs in a grid hosting environment," *BT Technology Journal*, vol. 22, no. 3, pp. 251–260, 2004.