GRIDTS: A New Approach for Fault-Tolerant Scheduling in Grid Computing

Fábio Favarim †* Joni da Silva Fraga†* Lau Cheuk Lung§* Miguel Correia‡
† Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil
§ Prog. de Pós-Graduação em Informática Aplicada, Pontifícia Universidade Católica do Paraná – Brazil
‡ LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
{*fabio, 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, on the contrary to traditional infrastructures where schedulers find resources for the tasks. This solution allows scheduling decisions to be made with up-to-date information about the resources, which is difficult in the traditional infrastructures. Moreover,* GRIDTS *provides fault-tolerant scheduling by combining a set of fault tolerance techniques to cope with crash faults in components of the system. The solution is mainly based a tuple space, which supports the scheduling and also provides support for the fault tolerance mechanisms.*

1. Introduction

The essence of grid computing is to provide efficient access to resources, maximizing the resource utilization while trying to minimize the job total execution time. The performance of the grid depends strongly on the efficiency of the scheduling. A schedule is an assignment of the tasks of a job to a set of resources. 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.

Many schedulers rely on accurate information about resources' attributes (CPU speed and load, memory) and tasks to do the scheduling. The information used by the schedulers is usually provided by an **information service** that is responsible for gathering data about all resources that compose the grid. Gathering this 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 This operation is reasonably costly in a large grid, and the snapshot tends to become outdated in a short time when the grid is comprised by a large number of non-dedicated, heterogeneous, widely-dispersed resources. The key problem is that information obtained from the information service may be outdated by the time the scheduler needs it to schedule tasks. Moreover, getting an accurate snapshot in an asynchronous distributed system (as the Internet) is impossible [7].

To overcome this complexity, we propose **GRIDTS**, a novel grid infrastructure. In our approach, resources select the tasks they want to execute, instead of the scheduler finding resources for the tasks. This solution does not use an information service and allows scheduling decisions to be made with up-to-date information, since, naturally, each resource has always up-to-date information about itself. In a nutshell, in our approach an user gives its job to a broker that breaks it in tasks and inserts these tasks in the tuple space. The available 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 from the tuple space, execute task...

GRIDTS is based on the generative coordination model, in which processes (brokers, resources) interact through a shared memory object called **tuple space** [10]. 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 [5]. This makes it particularly suited for highly dynamic systems like a grid.

In large-scale grids the probability of failures occurring 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, in the sense that all components in the system can fail by crashing and the system still behaves as expected.

Fault tolerance in GRIDTS 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

*Supported in part by CNPq.

is used to enforce the fault-tolerance and availability of the tuple space. In [8] we gave a brief overview of GridTS, focusing mainly on its experimental evaluation. This paper, on the contrary, makes a detailed presentation of the fault-tolerant scheduling algorithm.

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 – transactions, checkpointing, replication – to tolerate crash faults in any component of the system.

2. Tuple Spaces

The generative coordination model, introduced in the Linda programming language, allows distributed processes to interact through a shared memory object called **tuple space**, which can be implemented on a network using one or more servers [10]. In this space, generic data structures called *tuples* can be inserted, read and removed.

A tuple is an ordered sequence of typed fields. Given a tuple $t = \langle f_1, f_2, ..., f_n \rangle$, each field f_i can be: **actual**, i.e., a value; **formal**, i.e., a variable name preceded by a question mark ("?"); or a **wildcard**, like "*" meaning any value. A tuple in which all fields are actual is called an **entry** and is denoted by a lowercase letter, e.g., t. A tuple with at least one formal field is called a **template** and is denoted by $\bar{t}$. A tuple space can only store entries, never templates. Templates are used to read tuples in the space.

An important characteristic of the generative coordination model is the associative nature of the communication: tuples are not accessed through an address or identifier, but rather by their content. An entry t and a template $\bar{t}$ are said to **match** if : (i) both have the same number of fields; (ii) corresponding fields have the same type, and; (iii) corresponding actual fields have the same value. A tuple space provides three basic operations [11, 10]:

- $out(t)$: puts the tuple t in the tuple space (write);
- $in(\bar{t})$: reads and removes a tuple t that matches $\bar{t}$ from the tuple space. If no matching tuple t is available, the process stays blocked until a tuple matching the template $\bar{t}$ is available in the space (destructive read);
- $rd(\bar{t})$: has a behavior similar to in, but leaves the matched tuple t in the tuple space (non-destructive read).

A typical extension to this model, which we adopt in this paper, is the provision of non-blocking variants of the read operations: inp and rdp. These operations work in the same way as their blocking versions but return a "tuple not found" value if no matching tuple is available in the tuple space. All read operations are non-deterministic, because if there is more than one matching tuple available, one of them is chosen arbitrarily. In this paper we also use a variation of the rd operation that reads all the tuples that match the template, $copy\_collect(\bar{t})$ [17].

2.1. Fault tolerance

Fault tolerance in the generative coordination model has been considered in the literature from two points of view: (i) the construction of fault-tolerant tuple spaces using replication [21, 3]; and (ii) application-level fault tolerance mechanisms [13, 3]. The objectives of these works are essentially to guarantee that (i) the service provided by the tuple space stays available if there are crashes, by replicating the server in several computers, and (ii) the tuple space state stays valid according to the semantics of application when application processes crash, by using transactions. We use both to provide fault tolerance in our grid infrastructure.

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 essentially solved [21, 3, 15, 20].

Transactions

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. If the process executes all operations in the set, then the transaction is said to **commit**. If the process fails (crashes) during the execution, then the transaction is **aborted**; if some of the operations have already been executed then the tuple space removes all their effects to guarantee the atomicity [4].

In practice, the detection of a failure works the following way [20]. When a process starts a transaction, it defines a *lease* to do that transaction, i.e., a time interval during which it will execute the transaction's operations. If the process does not commit during that time, the tuple space assumes that the process failed and aborts the transaction. If the process needs more time to execute the transaction, it periodically renews the lease. This lease mechanism involves time assumptions about maximum processing and communication times.

In this paper, we consider that the tuple space provides two operations to indicate the beginning and termination of a transaction: $begintransaction$ and $committransaction$. The semantics of the transactions for each of the tuple space read/write operations is [13, 15, 20]:

- out: when a tuple is written inside a transaction, it is not accessible to operations outside of the transaction until the transaction commits. If the transaction aborts,

the tuples written inside the transaction are removed, leaving the space as if the operations never occurred.

- in/inp: a tuple removed from the space inside a transaction might have been written in the tuple space either inside or outside of the transaction. If the transaction aborts and a tuple written outside the transaction has been removed inside the transaction, this tuple is returned to the space leaving the space as if the removal never occurred. When a tuple is removed from tuple space inside a transaction, it is locked preventing other process from reading or removing it.
- rd/rdp: when a tuple is read from the space inside a transaction, it might have been written inside or outside the transaction. When a tuple is read from tuple space, it is locked preventing other processes from removing it.

Nested transactions

Some extensions of these **basic** transactions have been proposed to deal with specific applications' necessities. In this paper we use one of those extensions, **nested top-level transactions** [16]. A nested transaction is a transaction that starts and finishes inside the scope of another transaction. The transaction that starts the nested transaction is called the **parent**. A nested top-level transaction commits or aborts independently of the transaction that created it: if it commits the operations executed inside it take effect even if the parent transaction aborts; if it aborts it does not force the parent transaction to abort.

Due to the simplicity of tuple spaces and their language independence, current programming languages provide this communication and interaction model. Among them, the JavaSpaces [20] – part of Java's Jini framework – and the TSpaces [15] are some of the most known. Both support transactions.

3. GRIDTS

This section presents GRIDTS. We begin by presenting the system model and the properties that GRIDTS has to satisfy, then we give an overview of GRIDTS. Finally, we describe the algorithms executed by the brokers and the resources, and their correctness proofs.

3.1. System Model

The system is composed by an infinite set of **clients** and a set of n **servers** $U = \{s_1, s_2, ..., s_n\}$, which implement a fault-tolerant tuple space. Fault tolerance is enforced by replicating the tuple space in all servers [3, 21]. We assume that an arbitrary number of clients and up to f servers can be subject to **crash failures**. Each process behaves according to its specification until it possibly crashes or stops executing for some reason. A process that never crashes is said to be **correct**, while a process that crashes is said to be *faulty*. The client processes are divided in two subsets: the set $B = \{b_1, b_2, b_3, ...\}$ of **brokers** and the set $R = \{r_1, r_2, r_3...\}$ of **resources**. The client processes can only communicate through the tuple space.

We assume the **bag-of-tasks** computation model in which the tasks of an application are executed independently [18], therefore there is no communication between resources and no communication between the brokers. The grid can be composed by heterogeneous machines, in different administrative domains, and/or volunteer machines dispersed over the Internet.

We assume that the communication and processing are asynchronous, i.e., that there are no time bounds on the communication and processing delays. However, we also assume the existence of the transactional services introduced in Section 2.1, which cannot be implemented in a strictly asynchronous system (additional time assumptions are necessary).

It is important to clarify that the abnormal termination of a resource is not the only reason for a task to stop being executed. If a resource becomes unavailable to the grid for any reason, for instance, because its owner needs to use it, then we also consider it as a crash of the resource.

3.2. GRIDTS Properties

Distributed systems have been specified in terms of safety and liveness properties [1]. Informally, a safety property states that "something bad will *not* happen" during the system execution, while a liveness property states that eventually "something good *must* happen".

Consider that a *task ready to be executed* corresponds to a tuple describing the task on the tuple space. There are two properties that have to be satisfied by GRIDTS, one of safety and one of liveness:

- *Partial correctness*: if a resource executing a task fails, then the task becomes again ready to be executed.
- *Starvation freedom*: if there is some task ready to be executed and a correct resource able to execute it, then this task will eventually be executed.

The first property (safety) says that a task does not disappear if the resource that is executing it fails. The second property (liveness) says that every task will be executed if there is at least one correct resource capable of executing it, i.e., no task will be waiting forever to be executed. These are the main properties that the system has to satisfy to guarantee that all tasks are executed if there is at least one resource that does not fail.

3.3. Overview of GRIDTS

Resource allocation is an important problem in distributed systems, and the same is true in grid computing. In this context the problem is finding resources to execute the tasks that compose an application. Each resource is assumed to be available during a limited time and it is also assumed that typically there are not enough resources to execute all tasks. Scheduling tasks to the resources is usually made by schedulers that get the information about the available resources from a information service [9] and use this information to choose which resources will use to execute the tasks. This paper proposes a completely different scheduling scheme.

In GRIDTS, we use a tuple space for supporting task scheduling. Briefly, the idea is the following. Tuples describing the tasks to be executed are placed by the user in the tuple space. The grid resources retrieve from the tuple spaces tuples that describe tasks that they are able to execute, and execute them. After each execution, the result is placed in the tuple space, becoming available via broker to the user who submitted the tasks to the grid. Each task represents one unit of work that may be performed in parallel with other tasks. The description of a task contains all the necessary information for its execution, like: 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, and the parameters (input data) to the execution of the task. The users do not need to know what resources will execute the tasks, their location or when these resources will be available.

Therefore, our approach leverages naturally the full decentralization of the scheduling. Moreover, due to the flexibility of our infrastructure and the time and space decoupling of the involved entities, it is not necessary to have an information service because the resources themselves search for the tasks to be executed.

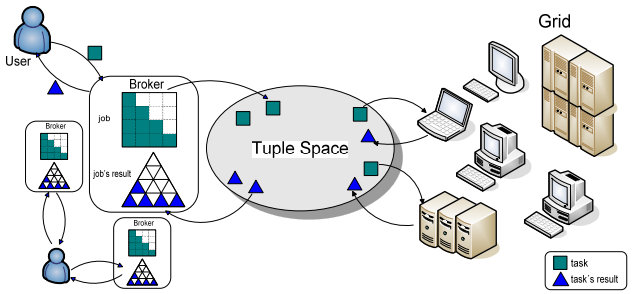

Figure 1. The GRIDTS infrastructure

Figure 1 presents the GRIDTS infrastructure. Scheduling of tasks is based on the *replicated-worker* pattern, also known as *master-workers* pattern [6]. 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 their **users**, divide the jobs in **tasks** and make these tasks available to the **resources** that compose the grid. Brokers are usually specific to one class of applications, i.e., they only know how to decompose jobs of this class. For example, if the application deals with processing satellite images, the broker decomposes the image (job) in several parts (tasks), that can be analyzed by different resources. When a resource/worker finishes executing a task, it gives the result to the broker that assembles all results and returns them to the user. All communication between brokers/masters and resources/workers is done exclusively through the tuple space.

The architecture of GRIDTS has the immediate benefit of not requiring an information service for indicating the resource utilization. On the contrary, it 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 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 resources' failures.

3.4. Designing GRIDTS

As presented in Section 3.3 the GRIDTS infrastructure comprises brokers, resources and the tuple space. In this section we define the behavior of the brokers and resources by presenting the algorithms they execute.

To facilitate the understanding of algorithms, the structure of the tuples used is described below. In all tuples, the first field is the name of the tuple. Most tuples contain some of the fields $jobId$ and $taskId$, which identify the job and the task, respectively:

- $\langle \text{“}TICKET\text{”}, ticket \rangle$ represents the tickets used to enforce an order on task execution. The objective is to guarantee the fairness of the scheduling, i.e., starvation freedom. The $ticket$ field contains the ticket number. The tuple space is initialized with a tuple $\langle \text{“}TICKET\text{”}, 0 \rangle$.
- $\langle \text{“}JOB\text{”}, jobId, nTasks, ticket, information, code \rangle$ represents all common information of tasks from the same job. The $nTasks$ field contains the number of tasks that compose the job, $ticket$ is the ticket value associated with the job, and $information$ indicates the attributes for job execution (e.g., the required processor speed, memory, operating system). The field $code$ can contain either the code to be executed or a reference to its location (e.g., an URL).
- $\langle \text{“}TASK\text{”}, jobId, taskId, information, parameters \rangle$ represents the task to be executed. The $information$ field has attributes for task execution. The $parameters$ field contains

input data for the task or a reference to its location. A task is uniquely identified by $jobId$ and $taskId$.

- $\langle$"$RESULT$", $jobId$, $taskId$, $result\rangle$ represents the result of a task execution. The $result$ field contains the result or a reference to its location.
- $\langle$"$CHECKPOINT$", $jobId$, $taskId$, $checkpoint\rangle$ represents the state of a task after a partial execution, i.e., a checkpoint. If a resource fails during the execution of a task, this checkpoint is used by another resource to continue executing the task. The $checkpoint$ field contains the partial state of task computation or a reference.
- $\langle$"$TRANS$", $transId$, $ticket$, $jobId\rangle$ indicates the last transaction executed by a broker, since brokers execute a sequence of two transactions. The objective is to prevent the broker from re-executing the same transaction if it fails and recovers. $transId$ identifies the last process transaction successfully committed. $ticket$ and $jobId$ have the usual meanings and are used to specify what the broker was doing when it failed.

There can be minor variations of the job and task tuples. The job and task tuples described above were designed for applications whose tasks execute the same code and only the input data is different for each task. Trivial modifications are needed, for instance, for applications whose tasks execute different code, but have the same input data, or for applications in which both the code and the input data is different for all tasks.

3.4.1 Broker

The algorithm executed by the brokers is presented in Algorithm 1. It uses (basic, not nested) transactions to tolerate brokers faults. A first transaction is used to ensure that the job tasks are inserted atomically in the space, i.e., either all of them are inserted or none (in case the broker fails during the insertion). The second transaction is used to get results of the job tasks atomically from the space. These transactions allow also the broker not to be locked waiting until all the tasks are executed, i.e., the broker can leave the system after having placed the tasks into the space and later run again to get the results.

The algorithm starts by verifying if the broker has been restarted due to a failure. This is done using the $rdp()$ operation (line 2). If this operation does not return a tuple then $transId = 1$ (line 1), the job is divided in a set of tasks (line 4) and the first transaction is executed (lines 5-14), possibly followed by the execution of the second transaction. Otherwise, line 2 sets $transId$ to the value of the third field of the tuple, which in this algorithm must be 2, and the second transaction is executed (lines 17-25). This second situation happens when the first transaction has been completely executed (so the out in line 13 was inserted in the tuple space) but the second transaction was interrupted due to a failure

Algorithm 1 Broker b_i

procedure broker($jobId, information, parameters, code$)

```
 1: transId = 1;
 2: rdp("TRANS", ?transId, ?ticket, jobId)
 3: if (transId = 1) then
 4:     tasks ← generateTasks(parameters);
 5:     begin transaction   // transaction 1- inserts tasks in tuple space
 6:         in("TICKET", ?ticket);
 7:         out("TICKET", ++ticket);
 8:         out("JOB", jobId, nTasks, ticket, information, code);
 9:         for i ← 1 to nTasks do
10:             out("TASK", jobId, tasks[i].id, task[i].information,
                                                tasks[i].parameters);
11:         end for
12:         transId = 2;
13:         out("TRANS", transId, ticket, jobId);
14:     commit transaction
15: end if
16: if (transId = 2) then
17:     begin transaction // transaction 2- gets results from tuple space
18:         for taskId ← 1 to nTasks do
19:             inp("RESULT", b_i, jobId, taskId, ?r);
20:             result ← result ∪ {r};
21:         end for
22:         in("TRANS", 2, ticket, jobId)
23:         in("JOB", jobId, nTasks, ticket, information, code);
24:         deliverToUser(result);
25:     commit transaction
26: end if
```

of the broker (so the in in line 22 was not executed). Therefore, the objective is to avoid the tasks from being reinserted in the tuple space due to a failure.

The first transaction starts by getting, incrementing and writing again the *ticket* tuple in the space (lines 6-7). These operations must be done inside a transaction because if the *ticket* tuple is removed from the space and not reinserted, then no more jobs can be inserted in the space. After handling the ticket, transaction 1 puts one *job* tuple describing the job in the space, and the corresponding *task* tuples (lines 8-11). Transaction 1 finishes with the insertion of the *trans* tuple in the space, indicating that this transaction was successfully executed (lines 12-13).

Transaction 2 gets the results of the tasks from the tuple space (lines 18-21). The *trans* tuple and the *job* tuple are removed from the space (lines 22-23). The result is delivered to the user in a reliable way (line 24).

3.4.2 Resource

Algorithm 2 describes the functioning of a resource r_i. The algorithm starts by searching all *job* tuples in the space ($copy\_collect$ operation - line 2) and choosing the most adequate to be executed according to some criteria – $chooseJob()$ function (line 3). To guarantee a fair scheduling, i.e., the *starvation freedom* property, the criteria should be to choose the job with smallest *ticket* that the resource can execute. However, other ways to choose a job are pos-

Algorithm 2 Resource r_i

```
procedure resource()
 1: loop
 2:    jobList ← copy_collect("JOB", *, *, *, *, *, *);
 3:    job ← chooseJob(jobList);
 4:    if (job ≠ ⊥) then
 5:       taskId ← chooseTask(job);
 6:       if (taskId ≠ ⊥) then
 7:          begin transaction                    // gets and executes a task
 8:             inp("TASK", job.jobId, taskId, *, ?parameters);
 9:             result ← executeTask(job.jobId, taskId,
                                     parameters, job.code);
10:             out("RESULT", job.jobId, taskId, result);
11:          commit transaction
12:       end if
13:    end if
14: end loop
function executeTask(jobId, taskId, parameters, code)
15: repeat
16:    begin transaction                        // partially executes a task
17:       inp("CHECKPOINT", jobId, taskId, ?checkpoint)
18:       partialResult, checkpoint, taskFinished ←
                 partialExecute(code, parameters, checkpoint);
19:       if not (taskFinished) then
20:          out("CHECKPOINT", jobId, taskId, checkpoint)
21:       end if
22:    commit transaction
23: until (taskFinished)
24: return partialResult
```

sible, e.g., 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 [2]. This stimulates the users to donate their idle resources to execute application of others users and minimizing the free-riding users – users that consume resources donated by others without donating any of their own. This criteria, however, does not guarantee starvation freedom.

After the job selection, the resource selects the task it can execute according to some specific heuristic ($chooseTask$ operation - line 5). It is important to clarity that the performance of scheduling depends strongly on the efficiency of the heuristic chosen. Thus, any scheduling heuristic that uses only local information about resources can be used in GRIDTS.

We propose a simple heuristic by classifying both jobs and resources in **classes**. Resources are classified in classes $\mathcal{CR} = \{r_1, \ldots, r_{nc}\}$ according to their speed. For instance, if they are classified in three classes ($nc = 3$), the first class can have resources until 1GHz, the second one resources from 1 to 3GHz, and the third one over 3GHz. Tasks are classified in classes $\mathcal{CT} = \{t_1, \ldots, t_{nc}\}$ according to their size. It is the broker that is responsible for classifying the tasks in classes, putting this information in the task tuple (in the $information$ field). The number of task and resource classes rc is the same and there is a correspondence between classes: class r_1 should include the slower resources and class t_1 the smaller tasks; class r_{nc} should include the faster resources and class t_{nc} the larger tasks.

Resources start getting tasks from the corresponding class of tasks, i.e., if a resource belongs to class r_i it gets a task from class t_i. When there are no more tasks of class t_i, it tries to get a task of class t_{i+1}; if there no tasks from that class, it tries to get from t_{i+2}, etc.; if there are no more from class t_{nc} them it starts trying to get tasks from class t_{i-1}, etc. If there are no more tasks, it means that all job tasks are being (or have been already) executed. By enforcing faster resources to execute larger tasks first, and slow resources to execute smaller tasks first, the probability of large tasks being scheduled to slow resources is reduced, and the job execution tends to terminate faster.

After a task is chosen a transaction begins (lines 7-11). The task chosen is removed from the tuple space (line 8), executed ($executeTask$ operation - line 9) and the result is inserted in the space (line 10). The transaction guarantees that these three operations are done atomically. If the resource fails during the transaction, the *task* tuple is returned to the space and will be eventually executed by another resource (or the same if it later recovers).

In a grid environment, with hundreds, thousands, or even tens of thousands of resources, joins, exists and failures of resources are frequent. 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 resource – a **checkpoint** – in stable storage [14]. If the resource fails, then another resource, or even the same in case it recovers, continues the execution of the task from that checkpoint.

The execution of a task is described in Algorithm 2, lines 15-24. GRIDTS uses the tuple space as stable storage, so when a resource is executing a task, it periodically inserts a *checkpoint* tuple in the space. Before the resource starts executing a task, it searches for a *checkpoint* tuple in the tuple space (line 17). If it exists the resource starts executing the task from this checkpoint onwards.

The task execution involves a *top-level nested transaction* in lines 16-22 (see Section 2.1). If the resource fails when executing this transaction, the two transactions in the algorithm are aborted. However, the *checkpoint* tuple inserted in the last committed top-level nested transaction (line 20) remains in the tuple space, instead of being removed due to the abortion of the parent transaction. This is the purpose of using a top-level nested transaction.

3.5. Correctness Proofs

This section makes an argument that GRIDTS satisfies the two properties in Section 3.2. We start by proving the following lemma:

Lemma 1 *If there is some task ready to be executed and a correct resource able to execute it, then some task will eventually be executed.*

Proof (sketch): The lemma states that there is "some task ready to be executed", which means there are at least two tuples in the space (T is the $taskId$ and J is its $jobId$):

$$\mathcal{T}_J = \langle \text{“}JOB\text{”}, J, nTasks_J, ticket_J, information_J, code_J \rangle$$

$$\mathcal{T}_T = \langle \text{“}TASK\text{”}, J, T, information_T, parameters_T \rangle$$

The lemma also states that there is a resource r that can execute T and is correct, i.e., that executes Algorithm 2 forever without stopping.

The proof is by contradiction. Assume r does not execute any task after some arbitrary instant t. This is only possible in two situations:

- r blocks at one of the lines 1 to 24. However, an inspection of the algorithm shows that the only line that might block is line 2 since $copy\_collect()$ is a blocking operation, but this cannot happen since there is at least one job tuple in the space, $\mathcal{T}_J$.
- r does not manage to get a task tuple from the space, which is not possible because there is at least one task tuple in the space, $\mathcal{T}_T$.

This is a contradiction, so some task will eventually be executed. □

The following theorems state that GRIDTS satisfies the two properties in Section 3.2:

Theorem 1 *If there is some task ready to be executed and a correct resource able to execute it, then this task will eventually be executed (Starvation freedom).*

Proof (sketch): This theorem is similar to the lemma above but the lemma states that any task is executed, while the theorem is about **this** task.

Let us consider, like in the previous lemma, that the job is described by the job tuple $\mathcal{T}_J$ and the task by the task tuple $\mathcal{T}_T$. The lemma proves that some task is executed. Obviously we can apply the lemma iteratively to show that infinite tasks are executed. What we have to prove is that task T is not left behind indefinitely. This is enforced by the $ticket$ mechanism.

The job of the task to be executed is selected in lines 2-3 by function $chooseJob()$ (Algorithm 2). In the text in Section 4.3.2 we stated that this function chooses the job with the smallest ticket value, say, $ticket_{J'}$. We are interested in the case that T has not yet been executed, therefore $ticket_{J'} \leq ticket_J$. There are two cases:

- if $ticket_{J'} = ticket_J$ then eventually the resource(s) will execute all tasks of J, including T (given the lemma).
- if $ticket_{J'} < ticket_J$ then eventually the resource(s) will execute all tasks of J' and of all jobs with ticket smaller than $ticket_J$. We end up with the previous case, so T is eventually executed, like we wanted to prove.

□

Theorem 2 *If a resource executing a task fails, then the task becomes again ready to be executed (Partial correctness).*

Proof (sketch): A resource r executes a task T inside the transaction in lines 7-11 (Algorithm 2). The theorem is only relevant after the task tuple $\mathcal{T}_T$ is removed from the space in line 8. If r fails, the semantics of the transaction for the inp operation is clear: $\mathcal{T}_T$ is returned to the tuple space, like we wanted to prove (Section 2.1). A checkpoint tuple may also be inserted in the space in line 20 and left in the space in case of failure, due to the semantics of top-level nested transactions. However this does not interfere with the fact that $\mathcal{T}_T$ is returned to the space, so task T becomes again ready to be executed, like we wanted to prove. □

4. Related Work

TaskSpaces is a framework for grid computing [19]. The paper claims that the framework is based on tuple spaces but this does not seem to be true: tuple spaces partially inspired the approach, but TaskSpaces ends up using a communication mechanism similar to message queues. TaskSpaces uses two "tuple space" instances, one called *task bag* for tasks, and another called *result bag* for results. The application sends the tasks to the task bag, and the task bag sends those tasks to the resources. After executing a task, the resource puts a result in the result bag, from which the results are taken by the users. TaskSpaces uses an event notification model where resources register with the task bag in order to receive tasks. If tasks being executed in different resources need to communicate, they exchange information about their IP address and ports through yet another tuple space instance, called communication bag. TaskSpaces is not fault-tolerant.

PLinda [13] and FT-Linda [3] are fault-tolerant extensions of the Linda language. PLinda uses a checkpoint mechanism to tolerate faults on the tuple space, and uses a transaction mechanism to allow processes to execute multiple tuple space operations atomically. FT-Linda assumes a set of replicated tuple spaces interconnected by a network supporting total order broadcast [12]. FT-Linda has a restricted form of transactions mechanism called atomic guarded statements (AGS). AGSs can execute multiple tuple space operations atomically, but do not allow computation between the operations. Both Plinda and FT-Linda use the replicated-worker pattern to exemplify the use of

their fault tolerance extensions in a cluster environment. In both, the execution of a task by a resource is done inside a transaction context (or AGS context). Thus, if a resource fails while executing a task, the task can be executed by another resource, but all processing executed is lost, because the state of the process is only saved after the transaction is committed. Our approach, also executes a task within a transaction context, but we use a checkpointing mechanism that allows a task to be resumed from the last checkpoint saved in case of failure. Moreover, these works do not use classes to improve the scheduling or deal with the problem of fairness in job execution.

5. Final remarks

The paper presents GRIDTS in detail, including the algorithms executed by the brokers and resources. GRIDTS is a decentralized and fault-tolerant grid infrastructure, in which the resources pick the tasks to execute, instead of using a centralized scheduler. The communication is made using a tuple space, benefiting from it being decoupled in time and space. GRIDTS combines different fault tolerance techniques – checkpointing, transactions, replication – to provide fault-tolerant scheduling. GRIDTS has been simulated and its performance compared with other grid infrastructures [8]. We plan to implement GRIDTS in the near future, possibly integrating it with OurGrid [2].

Acknowledgments

We thank Alysson Neves Bessani for several suggestions about this work and João Felipe Santos for his assistance with the simulations. This work was supported by CNPq (Brazilian National Research Council) through processes 550114/2005-0 and 506639/2004-5, and CAPES/GRICES (Project TISD).

References

[1] B. Alpern and F. Schneider. Defining Liveness. Technical report, Department of Computer Science, Cornell University, 1984.

[2] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Job Scheduling Strategies for Parallel Processing*, pages 61–86. Springer Verlag, 2003.

[3] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 06(3):287–302, 1995.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2):82–89, 2000.

[6] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions Computer Systems*, 3(1):63–75, 1985.

[8] F. Favarim, J. da Silva Fraga, L. C. Lung, M. Correia, and J. F. Santos. Exploiting tuple spaces to provide fault-tolerant scheduling on computational grids. In *10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing*, pages 403–411, 2007.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[10] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, 1985.

[11] D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *1st Annual ACM Symposium on Principles of Distributed Computing*, pages 10–18, 1982.

[12] V. Hadzilacos and S. Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report 94-1425, Department of Computer Science, Cornell University, New York - USA, 1994.

[13] K. Jeong and D. Shasha. Plinda 2.0: A transactional/checkpointing approach to fault-tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 96–105, 1994.

[14] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[15] T. J. Lehman, S. W. McLaughry, and P. Wycko. TSpaces: The next wave. In *32th Annual Hawaii International Conference on System Sciences (HICSS'99)*, 1999.

[16] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programing Languages and Systems*, 5(3):381–404, 1983.

[17] A. I. T. Rowstron and A. Wood. Solving the Linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[18] 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 *2nd International Euro-Par Conference (EURO-PAR'96)*, pages 487–495, 1996.

[19] H. D. Sterck, R. S. Markel, T. Phol, and U. Rde. A lightweight Java taskspaces framework for scientific computing on computational grids. In *ACM Symposium on Applied Computing (SAC'03)*, pages 1024–1030, 2003.

[20] Sun Microsystems. JavaSpaces service specification. Available in http://www.jini.org/wiki/JavaSpaces_Specification, 2003.

[21] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *19th Symposium on Fault-Tolerant Computing (FTCS'89)*, pages 199–206, 1989.