Efficient Parallel Application Execution on Opportunistic Desktop Grids

Francisco Silva[1], Fabio Kon[2], Daniel Batista[2], Alfredo Goldman[2], Fabio Costa[3] and Raphael Camargo[4]
[1]*Universidade Federal do Maranhão*
[2]*Universidade de São Paulo*
[3]*Universidade Federal de Goiás*
[4]*Universidade Federal do ABC*
Brazil

1. Introduction

The success of grid systems can be verified by the increasing number of middleware systems, actual production grids, and dedicated forums that appeared in recent years. The use of grid computing technology is increasing rapidly, reaching more scientific fields and encompassing a growing body of applications (Grandinetti, 2005; Wilkinson, 2009).

A grid might be seen as a way to interconnect clusters that is much more convenient than the construction of huge clusters. Another possible approach for conceiving a grid is the opportunistic use of workstations of regular users. The focus of an opportunistic grid middleware is not on the integration of dedicated computer clusters (e.g., Beowulf) or supercomputing resources, but on taking advantage of idle computing cycles of regular computers and workstations that can be spread across several administrative domains.

In a desktop grid, a large number of regular personal computers are integrated for executing large-scale distributed applications. The computing resources are heterogeneous in respect to their hardware and software configuration. Several network technologies can be used on the interconnection network, resulting in links with different capacities in respect to properties such as bandwidth, error rate, and communication latency. The computing resources can also be spread across several administrative domains. Nevertheless, from the user viewpoint, the computing system should be seen as a single integrated resource and be easy to use.

If the grid middleware follows an opportunistic approach, resources do not need to be dedicated for executing grid applications. The grid workload will coexist with local applications executions, submitted by the nodes regular users. The grid middleware must take advantage of idle computing cycles that arise from unused time frames of the workstations that comprise the grid. By leveraging the idle computing power of existing commodity workstations and connecting them to a grid infrastructure, the grid middleware allows a better utilization of existing computing resources and enables the execution of computationally-intensive parallel applications that would otherwise require expensive cluster or parallel machines.

Over the last decade, opportunistic desktop grid middleware developers have been constructing several approaches for allowing the execution of different application classes, such as: (a) sequential applications, where the task to be run is assigned to a single grid node; (b) parametric or bag-of-tasks applications, where several copies of a task are assigned to different grid nodes, each of them processing a subset of the input data independently and without exchanging data; (c) tightly coupled parallel applications, whose processes exchange data among themselves using message passing or shared memory abstractions.

Due to the heterogeneity, high scalability and dynamism of the execution environment, providing efficient support for application execution on opportunist grids comprises a major challenge for middleware developers, that must provide innovative solutions for addressing problems found in areas, such as:

Support for a variety of programming models, which enables the extension of the benefits of desktop grids to a larger array of application domains and communities, such as in scientific and enterprise computing, and including the ability to run legacy applications in an efficient and reliable way. Important programming models to consider include message-passing standards, such as MPI (MPI, 2009), BSP (Bisseling, 2004; Valiant, 1990), distributed objects, publish-subscribe, and mobile agents. In this chapter we will concentrate on the support for parallel application models, in particular MPI and BSP, but also pointing to the extensions of grid management middleware to support other programming models.

Resource management, which encompasses challenges such as how to efficiently monitor a large number of highly distributed computing resources belonging to multiple administrative domains. On opportunistic grids, this issue is even harder due to the dynamic nature of the execution environment, where nodes can join and leave the grid at any time due to the use of the non-dedicated machines by their regular (non-grid) users.

Application scheduling and execution management, which also includes monitoring, that must provide user-friendly mechanisms to execute applications in the grid environment, to control the execution of jobs, and to provide tools to collect application results and to generate reports about current and past situations. Application execution management should encompass all execution models supported by the middleware.

Fault tolerance, that comprises a major requirement for grid middleware as grid environments are highly prone to failures, a characteristic amplified on opportunistic grids due their dynamism and the use of non-dedicated machines, leading to a non-controlled computing environment. An efficient and scalable failure detection mechanism must be provided by the grid middleware, along with a means for automatic application execution recovery, without requiring human intervention.

In this chapter, we will provide a comprehensive description of reputable solutions found in the literature to circumvent the above described problems, emphasizing the approaches adopted in the InteGrade[1] (da Silva e Silva et al., 2010) middleware development, a multi-university effort to build a robust and flexible middleware for opportunistic grid computing. InteGrade's main goal is to be an opportunistic grid environment with support for tightly-coupled parallel applications. The next section gives an overview of grid application programming models and provides an introduction to the InteGrade grid middleware, discussing its support for executing parallel applications over a desktop grid platform.

[1] Homepage: `http://www.integrade.org.br`

Next, we will concentrate in two central issues in the development of an opportunistic grid infrastructure: application scheduling and execution management and fault tolerance.

2. Application programming models

A programming model is a necessary underlying feature of any computing system. Programming models provide well-defined constructs to build applications and are a key element to enable interoperation of application components developed by third parties. A programming model can be tied to a given programming language or it can be a higher-level abstraction layered on top of the language. In the latter case, different application components can be built using different programming languages, and the programming model, as long as properly implemented by a platform, serves as the logical bridge between them. In the heterogeneous environment of computing grids, the need for such high-level programming models is even more evident as there can be many different types of machine architecture and programming languages, all needing to be integrated as part of a seamless environment for distributed applications.

While it is largely acknowledged that no one-size-fits-all solution exists when it comes to programming models, one can argue that some programming models are best suited for particular kinds of problems than others (Lee & Talia, 2003). Considering that grid computing environments can be used to run different kinds of applications in different domains, such as e-science, finance, numeric simulation and, more generally, virtual organizations, it follows that having a variety of programming model to choose from may be an important factor. A number of well-known programming models have been investigated for grid computing, including, but not limited to, remote procedure calls (as in RPC and RMI), tuple spaces, publish-subscribe, message passing, and Web services, as well as enhancements of such models with non-functional properties such as fault tolerance, dependability and security (Lee & Talia, 2003). Note that a main emphasis of programming models for grid computing is on communication abstractions, which is due to the fact that interaction among distributed application components is a key issue for grid applications.

The InteGrade middleware offers a choice of programming models for computationally intensive distributed parallel applications, MPI (Message Passing Interface), and BSP (Bulk Synchronous Parallel) applications. It also offers support for sequential and bag-of-tasks applications. The remainder of this section presents the basic concepts of the InteGrade middleware and its support for parallel programming models.

2.1 Introduction to the InteGrade grid middleware

The basic architectural unit of an InteGrade grid is a cluster, a collection of machines usually connected by a local network. Clusters can be organized in a hierarchy, enabling the construction of grids with a large number of machines. Each cluster contains a *cluster manager* node that hosts InteGrade components responsible for managing cluster resources and for inter-cluster communication. Other cluster nodes are called *resource providers* and export part of their resources to the grid. They can be either shared with local users (e.g., secretaries using a word processor) or dedicated machines. The cluster manager node, containing InteGrade management components, must be a stable machine, usually a server, but not necessarily dedicated to InteGrade execution only. In case of a cluster manager failure, only its managed cluster machines will become unavailable. Figure 1 shows the basic components that enable application execution.

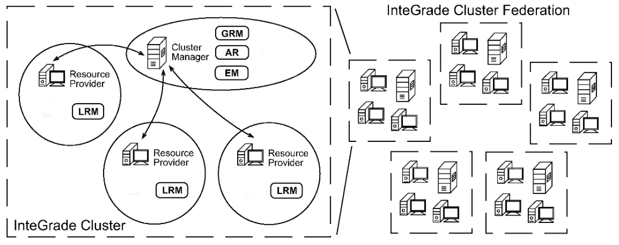

Fig. 1. InteGrade architecture.

Application Repository (AR): before being executed, an application must be previously registered with the Application Repository. This component stores the application description (metadata) and binary code.

Application Submission and Control Tool (ASCT): a graphical user interface that allows users to browse the content of the Application Repository, submit applications, and control their execution. Alternatively, applications can be submitted via the **InteGrade Grid Portal**, a Web interface similar to ASCT.

Local Resource Manager (LRM): a component that runs on each cluster node, collecting information about the state of resources such as memory, CPU, disk, and network. It is also responsible for instantiating and executing applications scheduled to the node.

Global Resource Manager (GRM): manages cluster resources by receiving notifications of resource usage from the LRMs in the cluster (through an information update protocol) and runs the scheduler that allocates tasks to nodes based on resource availability; it is also responsible for communication with GRMs in other clusters, allowing applications to be scheduled for execution in different clusters. Each cluster has a GRM and, collectively, the GRMs form the Global Resource Management service. We assume that the cluster manager node where the GRM is instantiated has a valid IP address and firewalls are configured to allow TCP traffic on the port used by the GRM. Network administrators establishing a Virtual Organization can, optionally, make use of ssh tunnels in order to circumvent firewalls and NAT boxes.

Execution Manager (EM): maintains information about each application submission, such as its state, executing node(s), input and output parameters, submission and termination timestamps. It also coordinates the recovery process in case of application failures.

Since grids are inherently more vulnerable to security threats than traditional systems, as they potentially encompass a large number of users, resources, and applications managed by different administrative domains, InteGrade encompass an opinion-based grid security model called Xenia. Xenia provides an authorization and authentication system and a security API that allows developers to access a security infrastructure that provides facilities such as digital signatures, cryptography, resource access control and access rights delegation. Using Xenia, we developed a secure Application Repository infrastructure, which provides authentication, secure communication, authorization, and application validation. A more detailed description of InteGrade security infrastructure can be found on de Ribamar Braga Pinheiro Júnior (2008); de Ribamar Braga Pinheiro Júnior et al. (2006).

2.2 InteGrade support for parallel applications

Executing computationally intensive parallel applications on dynamic heterogeneous environments, such as computational grids, is a daunting task. This is particularly true when using non-dedicated resources, as in the case of opportunistic computing, where one uses only the idle periods of the shared machines. In this scenario, the execution environment is typically highly dynamic, with resources periodically leaving and joining the grid. When a resource becomes unavailable, due to a failure or simply because the machine owner requests its use, the system needs to perform the necessary steps to restart the tasks on different machines. In the case of BSP or MPI parallel applications, the problem is even worse, since all processes that comprise the application may need to be restarted from a consistent distributed checkpoint.

2.2.1 InteGrade BSP applications

InteGrade's support for executing BSP applications adheres to the Oxford BSP API [2], targeted for the C language. Thus, an application based on the Oxford BSPlib can be executed over InteGrade with little or even no modification of its source code, requiring only its recompilation and linkage with the appropriate InteGrade libraries.

A BSP computation proceeds in a series of global supersteps. Each superstep comprises three ordered stages: (1) *concurrent computation*: computations take place on every participating process. Each process only uses values stored on its local memory. Computations are independent in the sense that they occur asynchronously of all others; (2) *communication*: at this stage, the processes exchange data between themselves; (3) *barrier synchronization*: when a process reaches this point (the barrier), it waits until all other processes have finished their communication actions. The synchronization barrier is the end of a superstep and the beginning of another one.

InteGrade's implementation of the BSP model uses CORBA (OMG, 2011) for inter-process communication. CORBA has the advantage of being an easier and cleaner communication environment, shortening development and maintenance time and facilitating system evolution. Also, since it is based on a binary protocol, the performance of CORBA-based communication is an order of magnitude faster than the performance of technologies based on XML, requiring less network bandwidth and processing power. On the shared grid machines, InteGrade uses OiL ((Maia et al., 2006)), a very light-weight version of a CORBA ORB that imposes a small memory footprint. Nevertheless, CORBA usage is completely transparent to the InteGrade application developer, who only uses the BSP interface (Goldchleger et al., 2005).

InteGrade's BSPLib associates to each process of a parallel application a *BspProxy*. The *BspProxy* is a CORBA servant responsible for receiving related communications from other processes, such as a virtual shared address read or write, or the receipt of messages signaling the end of the synchronization barrier. The creation of *BspProxies* is entirely handled by the library and is totally transparent to users.

The first created process of a parallel application is called *Process Zero*. Process Zero is responsible for assigning an unique identifier to each application process, broadcasting the

[2] The Oxford BSP Toolset `http://www.bsp-worldwide.org/implmnts/oxtool`

CORBA IORs of each process to allow them to communicate directly, and coordinating synchronization barriers. Moreover, *Process Zero* executes its normal computation on behalf of the parallel application.

On InteGrade, the synchronization barriers of the BSP model are used to store checkpoints during execution, since they provide global, consistent points for application recovery. In this way, in the case of failures, it is possible to recover application execution from a previous checkpoint, which can be stored in a distributed way as described in Section 4.3.1. Application recovery is also available for sequential, bag-of-tasks, and MPI applications.

2.2.2 Integrade MPI applications

Support for parallel applications based on MPI is achieved in InteGrade through MPICH-IG (Cardozo & Costa, 2008), which in turn is based on MPICH2[3], an open source implementation of the second version of the MPI standard, MPI2 (MPI, 1997). MPICH-IG adapts MPICH2 to use InteGrade's LRM and EM instead of the MPI daemon (MPD) to launch and manage MPI applications. It also uses the application repository to retrieve the binaries of MPI applications, which are dynamically deployed just prior to launch, instead of requiring them to be deployed in advance, as with MPICH2. MPI applications can thus be dispatched and managed in the same way as BSP or sequential applications.

In order to adapt MPICH2 to run on InteGrade, two of its interfaces were re-implemented: the *Channel Interface* (CI) and the *Process Management Interface* (PMI). The former is required to monitor a sockets channel to detect and treat failures. The latter is necessary to couple the management of MPI applications with InteGrade's Execution Manager (EM), adding functions for process location and synchronization.

Regarding communication among an application's tasks, MPICH-IG uses the MPICH2 Abstract Device Interface (ADI) to abstract away the details of the actual communications mechanisms, enabling higher layers of the communications infrastructure to be independent from them. In this way, we implemented two underlying communications channels: a CORBA-based one, for tasks running on different, possibly heterogeneous, networks, and a more efficient one, based on sockets, for tasks that reside in a single cluster.

Another feature of MPICH-IG, in contrast with conventional MPI platforms, refers to the recovery of individual application tasks after failures. This prevents the whole application from being restarted from scratch, thus contributing to reduce the makespan of application execution. Application recovery is supported by monitoring the execution state of tasks. Faulty tasks are then resumed on different grid nodes, selected by the GRM. Task recovery is implemented using system-level checkpoints. While other MPI platforms focus specifically on fault-tolerance and recovery, notably MPICH-V (Bosilca et al., 2002), they usually rely on homogeneous, dedicated, clusters. MPICH-IG removes this limitation to enable the dynamic scheduling of non-dedicated machines.

These features also favor MPICH-IG when compared to other approaches to integrate MPI into grid computing environments, such as MPICH-G2 (Karonis et al., 2003). In common to MPICH-G2 is the ability to run MPI applications on large scale heterogeneous environments, as well as the ability to switch from one communications protocol to another, depending on the relative location of the application tasks. However, MPICH-IG's ability to use

[3] http://www.mcs.anl.gov/research/projects/mpich2/

non-dedicated resources in an opportunistic way further contributes to scale up the amount of available resources. In addition, MPICH-IG enables legacy MPI applications to be transparently deployed on an InteGrade grid, without the need to modify their source code.

3. Application scheduling and execution management

Grid scheduling is a decision making process involving resources belonging to multiple administrative domains. As usual, this process includes a resource search for running applications. However, unlike traditional schedulers for distributed and parallel systems, grid schedulers have no control over the resources and applications in the system. Thus, it is necessary to have components that allow, among other features, resource discovery, monitoring and storage of information regarding resources and applications, mechanisms to allow access to different administrative domains and, depending on the adopted scheduling strategy, an approach for estimating the resources performance and the prediction of the applications execution times.

According to Schopf et al. (Schopf, 2004), a grid scheduling process can be broadly divided into three stages: filtering and resource discovery, resource selection, and preparing the environment for application execution. In the first stage, the grid scheduler creates a filter to select resources based on the restrictions and preferences provided by users during the application submission process. An information system usually provides a set of static and dynamic data about the available resources in the grid, such as their CPU capacity, the amount of available memory, and the network delay for delivering packets. Depending on the adopted scheduling heuristic, a cost estimator can sort the resources according to their efficiency to perform a certain type of code by using an analytic benchmark. In the second stage of the scheduling process, the scheduler will generate the applications mapping to resources in accordance with the system objectives, such as to minimize the response time of applications or to maximize the number of applications completed per time unit (throughput) (Dong & Akl, 2006; Zhu, 2003). In the third stage, a component running on the selected grid resource receives the application sent by the grid scheduler and prepares the environment for its execution by, for example, transferring the files containing the application input data.

A grid scheduling system can be organized in different schemes, according to different interests regarding performance and scalability (Subramani et al., 2002):

- Centralized: in this model, the scheduler maintains information about all administrative domains. All applications are submitted to the scheduler. Based on the queue of submitted applications and the information of all administrative domains, it makes the scheduling decisions.
- Hierarchical: in this scheme, each domain has its own local scheduler that are interconnected in a hierarchical structure. A request for an application execution that can not be handled with the locally available resources is sent up in the hierarchy, reaching an scheduler that has a broader view of the grid resources.
- Distributed: in this model, each domain has its own scheduler and the schedulers regularly consult each other to collect updated information about local loads. An application submitted for a given domain can then be transferred for execution in another domain that is less burdened.

3.1 Grid scheduling algorithms

A scheduling algorithm determines how the applications should be distributed for execution and how resources should be used, according to the system performance goals and the provided information. However, task mapping to a set of heterogeneous resources is a well known NP-complete problem (El-Rewini et al., 1994; Ibarra & Kim, 1977).

Scheduling algorithms can be grouped into two modes: batch and on-line (Maheswaran et al., 1999). In the on-line scheduling mode, a task is mapped to a machine as soon as it arrives to the scheduler. In the batch mode, tasks are not mapped as they arrive. Instead, they are placed inside a data structure called a meta-task and are mapped only in predefined interval times, called mapping events. In this way, batch algorithms can compare the task's resource requirements for performing better mappings.

Dong and Akl (Dong & Akl, 2006) describe scheduling algorithms commonly used in grid computing environments, among which we highlight the following:

Work Queue (WQ) is an on-line algorithm that assigns each task to the next machine that becomes available. If multiple machines become available simultaneously, one is randomly chosen. This algorithm maps only one task to a single resource. WQ does not use any specific information regarding the application or the grid resources and is particularly useful for systems where the main objective is to maximize the resources usage, instead of minimizing the execution time of individual tasks. Depending on the implementation, this algorithm may need to check the status of all m grid resources to find out which machines are available. Thus, the scheduling complexity of this algorithm is $O(m)$. An extension of this algorithm, called WQR, uses a replication mechanism. This variant is used by the OurGrid middleware [4].

Minimum Conclusion Time (MCT) is an on-line heuristic that assigns each task to the machine with the shorter expected completion time to accomplish it. The completion time corresponds to the sum of the time necessary for the machine to become available (in case it is already running another tasks) plus the time that it will take in order to execute the task. This algorithm can map more than one task per resource. The mapping complexity is $O(m)$, since as a task arrives, all grid machines are examined for determining the one having the shortest expected conclusion time for its execution.

Min-min is a batch heuristic based on MCT. This algorithm takes as input parameters a set of unmapped tasks M (meta-task) and a set of grid machines R. At its first step, the algorithm computes the completion time of each task in M for every machine in R. Next, the algorithm searches for the lowest completion time for each task. Min-min then selects the task with the minimum completion time among all tasks in M and assigns it to the machine in which this performance is expected to be obtained. The mapped task is removed from the meta-task M, and the algorithm increments the expected available time of the chosen grid resource considering the time to run the newly mapped task. This process is repeated until there are no more tasks left to be scheduled. As MCT, min-min also maps more than one task per node. Being m the number of tasks in M and r the amount of resources contained in R, computing the completion time of each task in all machines will take $O(mr)$. The loop is repeated m times, leading to a total cost of $O(m^2r)$.

[4] `http://www.ourgrid.org`

Task Grouping is an algorithm for scheduling applications comprising a large amount of short-duration tasks. In this case, scheduling and distributing each individual task would overload the system during the tasks transfer to grid resources. The algorithm groups the application's tasks according to their computation sizes and the processing power of the grid resources. Each group is sent to a single resource, reducing the required transmission overhead for task transferring.

Algorithms such as min-min and MCT need to compute how long it would take for running applications on grid resources. This information can be estimated by prediction algorithms (Liu, 2004), which usually follow two basic approaches. The first approach calculates an application estimated execution time based on stored records of previous runs of the same or similar applications. The second approach is based on knowledge regarding the application execution model, which are usually parallel applications with divisible workloads (MPI or PVM). The application code is analyzed, estimating the execution time of each task according to the capacity of the grid resources. An example of this latter approach is the PACE (Performance Analysis and Characterization Environment).

3.2 InteGrade application scheduling

The InteGrade scheduling algorithm (Goldchleger et al., 2004) follows an on-line approach. It uses a filter to select resources based on constraints and preferences provided by users during the process of submitting applications. Constraints define minimum requirements for the selection of machines, such as hardware and software platforms, resource requirements such as minimum memory requirements. Preferences define the order used for choosing the resources, like rather executing on a faster CPU than on a slower one. The tasks that make up an application are then mapped to the nodes according to the ordered list of resources. If requirements and preferences are not specified, the algorithm maps the tasks to random chosen grid resources. The algorithm can map more than one task per node.

3.2.1 InteGrade resource availability prediction

The success of an opportunistic grid depends on a good scheduler. An idle machine is available for grid processing, but whenever its local users need their resources back, grid applications executing at that machine must either migrate to another grid machine or abort and possibly restart at another node. In both cases, there is considerable loss of efficiency for grid applications. A solution is to avoid such interruptions by scheduling grid tasks on machines that are expected to remain idle for the duration of the task.

InteGrade predicts each machine's idle periods by locally performing Use Pattern Analysis of machine resources at each machine on the grid, as described in Finger et al. (2008; 2010). Currently, four different types resource are monitored: CPU use, RAM availability, disk space, and swap space.

Use pattern analysis deals with *machine resource use objects*. Each object is a vector of values representing the time series of a machine's resource use. The sampling of a machine's resource use is performed at a fixed rate (once every 5 minutes) and grouped into objects covering 48 hours with a 24-hour overlap between consecutive objects. InteGrade employ 48-hour long objects so as to have enough past information to be used in the runtime prediction phase.

The Use Pattern Analysis performs unsupervised machine learning (Barlow, 1999; Theodoridis & Koutroumba, 2003) to obtain a fixed number of *use classes*, where each class is represented by its *prototypical* object. The idea is that each class represents a frequent use pattern, such as a busy work day, a light work day or a holiday. As in most machine learning processes, there are two phases involved in the process, which in the InteGrade architecture are implemented by a module called Local Use Pattern Analyzer (LUPA), as follows.

The Learning Phase. Learning is performed off-line, using 60 objects collected by LUPA during the machine regular use. A clustering algorithm (Everitt et al., 2001) is applied to the training data, such that each cluster corresponds to a use class, represented by a prototypical object, which is obtained by averaging over the elements of the class. Learning can occur only when there is a considerable mass of data. InteGrade approach requires at least two months of data. As data collection proceeds, more data and more representative classes are obtained.

The Decision Phase. There is one LUPA module per machine on the grid. Requests are sent by the scheduler specifying the amount of resources (CPU, disk space, RAM, etc.) and the expected duration needed by an application to be executed at that machine. The LUPA module decides whether this machine will be available for the expected duration, as explained below. LUPA is constantly keeping track of the current use of resources. For each resource, it focuses on the recent history, usually the last 24 hours, and computes a distance between the recent history and each of the use classes learned during the training phase. This distance takes into account the time of the day in which the request was made, so that the recent history is compared to the corresponding times in the use classes. The class with the smallest distance is the *current use class*, which is used to predict the availability in the near future. If all resources are predicted to be available, then the application is scheduled to be executed; otherwise, it is rejected.

3.3 Application management: a mobile agents approach

In distributed systems such as opportunistic grids, failures can occur due to several factors, most of them related to resource heterogeneity and distribution. These failures together with the use of the resources by its owners modify grid resource availability (i.e. resources can be active, busy, off-line, crashed, etc.). An opportunistic grid middleware should be able to monitor and detect such changes, rescheduling applications across the available resources, and dynamically tuning the fault tolerance mechanisms to better adapt to the execution environment.

When dealing with bag-of-tasks like applications, one interesting approach may be the use of mobile agents (Pham & Karmouch, 1998), which allows the implementation of dynamic fault tolerance mechanisms based on task replication and checkpointing. A task replica is a copy of the application binary that runs independently of the other copies. Through these mechanisms a middleware may be capable of migrating tasks when nodes fail and coordinate task replicas and its checkpoints in a rational manner, keeping only the most advanced checkpoint and by migrating slow replicas. This dynamically improves application execution, compensates the misspent of resources introduced by the task replication and solves scalability issues.

These mechanisms compose a feedback control system (Goel et al., 1999; Steere et al., 1999), gathering and analyzing information about the execution progress and adjusting its behavior accordingly. Agents are suitable for opportunistic environments due to intrinsic characteristics such as:

1. *Cooperation*: agents have the ability to interact and cooperate with other agents; this can be explored for the development of complex communication mechanisms among grid nodes;
2. *Autonomy*: agents are autonomous entities, meaning that their execution goes on without any or with little intervention by the clients that started them. This is an adequate model for submission and execution of grid applications;
3. *Heterogeneity*: several mobile agent platforms can be executed in heterogeneous environments, an important characteristic for better use of computational resources among multi-organization environments;
4. *Reactivity*: agents can react to external events, such as variations on resources availability;
5. *Mobility*: mobile agents can migrate from one node to another, moving part of the computation being executed, helping to balance the load on grid nodes;

The InteGrade research group has been investigating the use of the agent paradigm for developing a grid software infrastructure since 2004, leading to the MobiGrid (Barbosa & Goldman, 2004; Pinheiro et al., 2011) and MAG (Mobile Agents for Grids) (Lopes et al., 2005) projects that are based on the InteGrade middleware.

4. Application execution fault-tolerance

On opportunistic grids, application execution can fail due to several reasons. System failures can result not only from an error on a single component but also from the usually complex interactions between the several grid components that comprise a range of different services. In addition to that, grid environments are extremely dynamic, with components joining and leaving the system at all times. Also, the likelihood of errors occurring during the execution of an application is exacerbated by the fact that many grid applications will perform long tasks that may require several days of computation.

To provide the necessary fault tolerance functionality for grid environments, several services must be available, such as: (a) **failure detection:** grid nodes and applications must be constantly monitored by a failure detection service; (b) **application failure handling:** various failure handling strategies can be employed in grid environments to ensure the continuity of application execution; and (c) **stable storage:** execution states that allow recovering the pre-failure state of applications must be saved in a data repository that can survive grid node failures. Those basic services will be discussed on the following sections.

4.1 Failure detection

Failure detection is a very important service for large-scale opportunistic grids. The high rate of churn makes failures a frequent event and the capability of the grid infrastructure to efficiently deal with them has a direct impact on its ability to make progress. Hence, failed nodes should be detected quickly and the monitoring network should itself be reliable, so as to ensure that a node failure does not go undetected. At the same time, due to the scale and geographic dispersion of grid nodes, failure detectors should be capable of disseminating information about failed nodes as fast and reliably as possible and work correctly even when no process has a globally consistent view of the system. Moreover, the non-dedicated nature of opportunistic grids requires that solutions for failure detection be very lightweight in terms of network bandwidth consumption and usage of memory and CPU cycles of resource provider machines. Besides all of these requirements pertaining to the functioning of failure detectors, they must also be easy to set-up and use; otherwise they might be a source of design and configuration errors. It is well-known that configuration errors are a common cause of grid failures (Medeiros et al., 2003). The aforementioned requirements are hard to meet as a whole and, to the best of our knowledge, no existing work in the literature addresses all of them. This is not surprising, as some goals, e.g., a reliable monitoring network and low network bandwidth consumption, are inherently conflicting. Nevertheless, they are all real issues that appear in large-scale opportunistic grids, and reliable grid applications are expected to deal with them in a realistic setting.

4.1.1 InteGrade failure detection

The InteGrade failure detection service (Filho et al., 2008) includes a number of features that, when combined and appropriately tuned, address all the above challenges while adopting reasonable compromises for the ones that conflict. The most noteworthy features of the proposed failure detector are the following: (i) a gossip- or infection-style approach (van Renesse et al., 1998), meaning that the network load imposed by the failure detector scales well with the number of processes in the network and that the monitoring network is highly reliable and descentralized; (ii) self-adaptation and self-organization in the face of changing network conditions; (iii) a crash-recover failure model, instead of simple crash; (iv) ease of use and configuration; (v) low resource consumption (memory, CPU cycles, and network bandwidth).

InteGrade's failure detection service is completely decentralized and runs on every grid node. Each process in the monitoring network established by the failure detection service is monitored by K other processes, where K is an administrator-defined parameter. This means that for a process failure to go undetected, all the K processes monitoring it would need to fail at the same time. A process r which is monitored by a process s has an open TCP connection with it through which it sends heartbeat messages and other kinds of information. If r perceives that it is being monitored by more than K processes, it can cancel the monitoring relationship with one or more randomly chosen processes. If it is monitored by more than K processes, it can select one or more random processes to start monitoring it. This approach yields considerable gains in reliability (Filho et al., 2009) at a very low cost in terms of extra control messages.

InteGrade's failure detector automatically adapts to changing network conditions. Instead of using a fixed timeout to determine the failure of a process, it continuously outputs the probability that a process has failed based on the inter-arrival times of the last W heartbeats and the time elapsed since the last heartbeat was received, where W is an administrator-defined parameter. The failure detector can then be configured to take recovery actions whenever the failure probability reaches a certain threshold. Multiple thresholds can be set, each one triggering a different recovery action, depending on the application requirements.

InteGrade employs a reactive and explicit approach to disseminate information about failed processes. This means that once a process learns about a new failure it automatically sends this information to J randomly-chosen processes that it monitors or that monitor it. The administrator-defined parameter J dictates the speed of dissemination. According to Ganesh et al. (2003), for a system with N processes, if each process disseminates a piece of information to $(\log n) + c$ randomly chosen processes, the probability that the information does not reach every process in the system is $e^{(-e^{(-c)})}$, with $n \rightarrow \infty$. For $J = 7$, this probability is less than 0.001. On the other hand, no explicit action is taken to disseminate information about new processes. Instead, processes get to know about new processes by simply receiving heartbeat messages. Each heartbeat that a process p sends to a process q includes some randomly chosen ids of K processes that p knows about. In a grid, it is important to quickly disseminate information about failed processes in order to initiate recovery as soon as possible and, when recovery is not possible and the application has to be re-initiated, to avoid wasting grid resources. On the other hand, information about new members is not so urgent, since not knowing about new members in general does not keep grid applications from making process.

InteGrade's group membership service is implemented in Lua (Ierusalimschy et al., 1996), an extensible and lightweight programming language. Lua makes it easy to use the proposed service from programs written in other programming languages, such as Java, C, and C++. Moreover, it executes in several platforms. Currently, we have successfully run the failure detector in Windows XP, Mac OS X, and several flavors of Linux. The entire implementation of the group membership service comprises approximately 80Kb of Lua source code, including comments.

4.2 Application failure handling

The main techniques used to provide application execution fault-tolerance can be divided in 2 levels: task level and workflow level (Hwang & Kesselman, 2003). At the task level, fault-tolerance techniques apply recovery mechanisms directly at the tasks, to masks the failures effects. At the workflow level, the recovery mechanisms creates recovery procedures directly in the workflow execution control.

4.2.1 Task-level techniques

There are 3 task-level techniques are frequently used in computational grids: retrying, replication and checkpointing.

Retrying is the simplest technique and consists in restarting the execution of the task after a failure. The task can be schedule at the same resource or at another one. Several scheduling can be used, such as FIFO (First-In First-Out), or algorithms that select resources with more computational power, more idleness or credibility (regarding security).

Replication consists in executing several replicas of the task on different resources, with the expectation that at least one of them finishes the execution successfully. The replicas can be scheduled to machines from the same or from different domains. Since networks failures can make an entire domain inaccessible, executing replicas in different domains improves the reliability. Replication is also useful to guarantee the integrity of the task execution results, since defective or malicious nodes can produce erroneous results. To prevent these errors, it is possible to wait until all executing tasks finish and apply a Byzantine agreement algorithm to compare the results.

Checkpointing consists in periodically store the application state in a way that, after a failure, the application can be restarted and continue its execution from the last saved state. The checkpointing mechanisms can be classified on how the application stated is obtained. There are two main approaches, called system-level and application-level checkpointing.

When using system-level checkpointing, the application state is obtained directly from the process memory space, together with some register values and state information from the operating system (Litzkow et al., 1997; Plank et al., 1998). This may require modifications in the kernel of the operating system, which may not be possible due to security reasons, but has the advantage that the checkpointing process can be transparent to the application. Some implementations permits the applications to be written in several program languages and used without recompilation. An important disadvantage of this approach for computational grids is that the checkpoints are not portable and is useful only in homogeneous clusters.

With application-level checkpointing, the application provides the data that will be stored in the checkpoint. It is necessary to instrument the application source-code so that the application saves its state periodically and, during recovery, reconstruct its original state from the checkpoint data (Bronevetsky et al., 2003; de Camargo et al., 2005; Karablieh et al., 2001; Strumpen & Ramkumar, 1996). Manually inserting code to save and recover an application state is a very error prone process, but this problem can be solved by providing a precompiler which automatically inserts the required code. Other drawbacks of this approach are the need to have access to the application source-code and that the checkpoints can be generated only at specified points in the execution. But this approach has the advantage that semantic information about memory contents is available and, consequently, only the data necessary to recover the application state needs to be saved. Moreover, the semantic information permits the generation of portable checkpoints (de Camargo et al., 2005; Karablieh et al., 2001; Strumpen & Ramkumar, 1996), which is an important advantage for heterogeneous grids.

In the case of coupled parallel applications, the tasks may exchange messages, which can be in transit during the generation of local checkpoints by the application processes. The content of these messages, including sending and delivery ordering, must also be considered as part of the application state. To guarantee that the global state (which includes the local state of all tasks) is consistent, it is necessary to use checkpointing protocols, which are classified as non-coordinated, coordinated and communication induced (Elnozahy et al., 2002).

The parallel checkpointing protocols differ in the level of coordination among the processes, from the non-coordinated protocols, where each process of the parallel application generates its local checkpoint independently from the others, to fully coordinated protocols, where all the processes synchronize before generating their local checkpoints. The communication induced protocol is similar to the non-coordinated one, with the difference that, to guarantee the generation of global checkpoints with consistent states, processes may be required to generate additional checkpoints after receiving or before sending messages.

4.2.2 Workflow-level techniques

Workflow-level techniques are based on the knowledge of the execution context of the tasks and on the flow control of the computations. They are applied to grids that support workflow-based applications and the more common techniques are: alternative task, redundancy, user defined exception handling and rescue workflow.

The basic idea of the *alternative task* technique consists in using a different implementation of the task to substitute the failed one. It is useful when a task has several implementations with distinct characteristics, such as the efficiency and reliability.

The *redundancy* technique is similar to the alternative task, with the difference that the distinct implementations of the task are executed simultaneously. When the first one finishes, the task is considered complete, and the remaining tasks are killed.

The *user defined exception handling* allows the user to provide the handling procedure for specific failures of particular tasks. This approach will usually be more efficient, since the user normally has specific knowledge about the tasks.

Finally, the last approach consists in generating a new workflow, called *rescue workflow*, to execute the tasks of the original flow that failed and the ones that could not be executed due to task dependencies. If the rescue workflow fails, a new workflow can be generated (Condor_Team, 2004).

4.2.3 InteGrade application failure handling

InteGrade provides a task-level fault tolerance mechanism. In order to overcome application execution failures, this mechanism provides support for the most used failure handling strategies: (1) **retrying**: when an application execution fails, it is restarted from scratch; (2) **replication**: the same application is submitted for execution multiple times, generating various application replicas; all replicas are active and execute the same code with the same input parameters at different nodes; and (3) **checkpointing**: periodically saves the process' state in stable storage during the failure-free execution time. Upon a failure, the process restarts from the latest available saved checkpoint, thereby reducing the amount of lost computation. As part of the application submission process, users can select the desired technique to be applied in case of failure. These techniques can also be combined resulting in four more elaborate failure handling techniques: *retrying* (without checkpoint and replication), *checkpointing* (without replication), *replication* (without checkpointing), and *replication with checkpointing*.

InteGrade includes a portable application-level checkpointing mechanism (de Camargo et al., 2005) for sequential, bag-of-tasks, and BSP parallel applications written in C. This portability allows an application's stored state to be recovered on a machine with a different architecture from the one where the checkpoint was generated. A precompiler inserts, into the application code, the statements responsible for gathering and restoring the application state from the checkpoint. On BSP applications, checkpoints are generated immediately after the end of a BSP synchronization phase. For MPI parallel applications, we provide a system-level checkpointing mechanism based on a coordinated protocol (Cardozo & Costa, 2008). For storing the checkpointing data, InteGrade uses a distributed data storage system, called OppStore (Section 4.3.1).

InteGrade allows replication for sequential, bag-of-tasks, MPI and BSP applications. The amount of generated replicas is currently defined during the application execution request, issued through the Application Submission and Control Tool (ASCT). The request is forwarded to the Global Resource Manager (GRM), which runs a scheduling algorithm that guarantees that all replicas will be assigned to different nodes. Another InteGrade component, called Application Replication Manager (ARM), concentrates most of the code responsible for managing replication. In case of a replica failure, the ARM starts its recovery process. When the first application replica concludes its job, the ARM kills the remaining ones, releasing the allocated grid resources.

4.3 Stable storage

To guarantee the continuity of application execution and prevent loss of data in case of failures, it is necessary to store the periodical checkpoints and application temporary, input, and output data using a reliable and fault-tolerant storage device or service, which is called stable storage. The reliability is provided by the usage of data redundancy and the system can determine the level of redundancy depending on the unavailability rate of the storage devices used by the service.

A commonly used strategy for data storage in computational grids usually store several replicas of files in dedicated servers managed by replica management systems (Cai et al., 2004; Chervenak et al., 2004; Ripeanu & Foster, 2002). These systems usually target high-performance computing platforms, with applications that require very large amounts (petabytes) of data and run on supercomputers connected by specialized high-speed networks.

When dealing with the storage of checkpoints of parallel applications in opportunistic grids, a common strategy is to use the grid machines used to execute applications to store checkpoints. It is possible to distribute the data over the nodes executing the parallel application that generates the checkpoints, in addition to other grid machines. In this case, data is transfered in a parallel way to the machines. To ensure fault-tolerance, data must be coded and stored in a redundant way, and the data coding strategy must be selected considering its scalability, computational cost, and fault-tolerance level. The main techniques used are *data replication*, *data parity*, and *erasure codes*.

Using data replication, the system stores full replicas of the generated checkpoints. If one of the replicas becomes unaccessible, the system can easily find another. The advantage is that no extra coding is necessary, but the disadvantage is that this approach requires the transfer and storage of large amounts of data. For instance, to guarantee safety against a single failure, it is necessary to save two copies of the checkpoint, which can generate too much local network traffic, possibly compromising the execution of the parallel application. A possible approach to adopt is to store a copy of the checkpoint locally and another remotely (de Camargo et al., 2006). Although a failure in a machine running the application makes one of the checkpoints unaccessible, it is still possible to retrieve the other copy. Moreover, the other application processes can use their local checkpoint copies. Consequently, this storage mode permits recovery as long as one of the two nodes containing a checkpoint replica is available.

The two other coding techniques decompose a file into smaller data segments, called stripes, and distribute these stripes among the machines. To ensure fault-tolerancy, redundant stripes are also generated and stored, permitting the original file to be recovered even if a subset of the stripes is lost. There are several algorithms to code the file into redundant fragments. A commonly used one is the use of data parity (Malluhi & Johnston, 1998; Plank et al., 1998; Sobe, 2003), where one or more extra stripes are generated based on the evaluation of the parity of the bits in the original fragments. It has the advantage that it is fast to evaluate and that the original stripes can be stored without modifications. But they have the disadvantage that data cannot be recovered if two or more fragments are lost and, consequently, cannot be used for storage in devices with higher rates of unavailability.

The other strategy is to use erasure coding techniques, which allow one to code a vector U of size n, into $m + k$ encoded vectors of size n/m, with the property that one can regenerate U

using only m of the $m + k$ encoded vectors. By using this encoding, one can achieve different levels of fault-tolerance by tuning the values of m and k. In practice, it is possible to tolerate k failures with an overhead of only $k/m * n$ elements. The information dispersal algorithm (IDA) (de Camargo et al., 2006; Malluhi & Johnston, 1998; Rabin, 1989) is an example of erasure code that can be used to code data. IDA provides the desired degree of fault-tolerance with lower space overhead, but it incurs a computational cost for coding the data and an extra latency for transferring the fragments from multiple nodes. But analytical studies (Rodrigues & Liskov, 2005; Weatherspoon & Kubiatowicz, 2002) show that, for a given redundancy level, data stored using erasure coding has a mean availability several times higher than using replication.

The checkpointing overhead in the execution time of parallel applications when using erasure coding, data parity and replication was compared elsewhere (de Camargo et al., 2006). The replication strategy had the smallest overhead, but uses more storage space. Erasure coding causes a larger overhead, but uses less storage space and is more flexible, allowing the system to select the desired level of fault-tolerance.

4.3.1 InteGrade stable storage

InteGrade implements a distributed data repository called OppStore (de Camargo & Kon, 2007). It is used for storing the application's input and output files and checkpointing data. Access to this distributed repository is performed through a library called *access broker*, which interacts with OppStore.

OppStore is a middleware that provides reliable distributed data storage using free disk space from shared grid machines. The goal is to use this free disk space in an opportunistic way, i.e., only during the idle periods of the machines. The system is structured as a federation of clusters and is connected by a Pastry peer-to-peer network (Rowstron & Druschel, 2001) in a scalable and fault-tolerant way. This federation structure allows the system to disperse application data throughout the grid. During storage, the system slices the data into several redundant, encoded fragments and stores them in different grid clusters. This distribution improves data availability and fault-tolerance, since fragments are located in geographically dispersed clusters. When performing data retrieval, applications can simultaneously download file fragments stored in the highest bandwidth clusters, enabling efficient data retrieval. This is OppSore standard storage mode, called perennial. Using OppStore, application input and output files can be obtained from any node in the system. Consequently, after a failure, restarting of an application execution in another machine can be easily performed. Also, when an application execution finishes, the output files uploaded to the distributed repositories can be accessed by the user from any machine connected to the grid.

OppStore also has an ephemeral mode, where data is stored in the machines of the same cluster where the request was issued. It is used for data that requires high bandwidth and only needs to be available for a few hours. It is used to store checkpointing (de Camargo et al., 2006; Elnozahy et al., 2002) data and other temporary application data. In this storage mode, the system stores the data only in the local cluster and can use IDA or data replication to provide fault-tolerance. For checkpointing data, the preferred strategy is to store two copies of each local checkpoint in the cluster, one in the machine where it was generated and the other on another cluster machine. During the recovery of failed parallel application, most of the processes will be able to recover their local checkpoints from the local machine.

The system stores most of the generated checkpoints using the ephemeral mode, since the usage of local networks generates a lower overhead. To prevent losing the entire computation due to a failure or disconnection of the cluster where the application was executing, periodically, the checkpoints are also stored in other clusters using the perennial storage mode, for instance, after every k generated checkpoints. With this strategy, InteGrade can obtain low overhead and high availability at the same time for checkpoint storage.

5. Conclusion

Corporations and universities typically have hundreds or thousands of desktop machines, which are used by workers as their personal workstations or by students in instructional and research laboratories. When analyzing the usage of each of these machines one concludes that they sit idle for a significant amount of time. Even when the computer is in use, it normally has a large portion of idle resources. When we consider the night period, we conclude that most of the times they are not used at all. This situation lives in contradiction with the growing demand for highly intensive computational power required by several fields of research, including physics, chemistry, biology, and economics. Several corporations also rely on the intensive use of computing power to solve problems such as financial market simulations and studies for accurate oil well drilling. The movie industry makes intensive use of computers to render movies using an increasing number of special effects.

Opportunistic grid middleware enables the use of the existing computing infrastructure available in laboratories and offices in universities, research institutes, and companies to execute computationally intensive parallel applications. Nevertheless, executing this class of applications on such a dynamic and heterogeneous environment is a daunting task, especially when non-dedicated resources are used, as in the case of opportunistic computing. In an opportunistic grid middleware resources do not need to be dedicated for executing grid applications. The grid workload coexist with local applications executions, submitted by the nodes regular users. The middleware must take advantage of idle computing cycles that arise from unused time frames of the workstations that comprise the grid.

An opportunistic grid middleware must provide innovative solutions to circumvent problems arising from the heterogeneity, high scalability and dynamism of the execution environment. Among the several aspects that must be considered, central issues are related to the support for a variety of programming models, highly scalable distributed resource management, application scheduling and execution management, and fault tolerance, since opportunist grid environments are inherently prone of errors. In this chapter, we provided a comprehensive description of reputable solutions found in the literature to circumvent the above described problems, emphasizing the approaches adopted in the InteGrade middleware, an open-source multi-university effort to build a robust and flexible middleware for opportunistic grid computing available at `http://www.integrade.org.br`.

6. References

Barbosa, R. M. & Goldman, A. (2004). Framework for mobile agents on computer grid environments, *First International Workshop on Mobility Aware Technologies and Applications (MATA 2004)*, Springer LNCS No 3284, Florianópolis, Brazil, pp. 147–157.

Barlow, H. B. (1999). *Unsupervised learning: Foundations of Neural Computation*, MIT Press.

Bisseling, R. H. (2004). *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press.

Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V. & Selikhov, A. (2002). Mpich-v: toward a scalable fault tolerant mpi for volatile nodes, *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society Press, Baltimore, Maryland, USA, pp. 1–18.

Bronevetsky, G., Marques, D., Pingali, K. & Stodghill, P. (2003). Automated application-level checkpointing of MPI programs, *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 84–89.

Cai, M., Chervenak, A. & Frank, M. (2004). A peer-to-peer replica location service based on a distributed hash table, *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, p. 56.

Cardozo, M. C. & Costa, F. M. (2008). Mpi support on opportunistic grids based on the integrade middleware, *2nd Latin American Grid International Workshop (LAGrid)*, Campo Grande, Brazil.

Chervenak, A. L., Palavalli, N., Bharathi, S., Kesselman, C. & Schwartzkopf, R. (2004). Performance and scalability of a replica location service, *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, IEEE Computer Society, Washington, DC, USA, pp. 182–191.

Condor_Team (2004). *Online Manual of Condor Version 7.4.4*, University of Wisconsin-Madison, http://www.cs.wisc.edu/condor/manual/v7.4.

da Silva e Silva, F. J., Kon, F., Goldman, A., Finger, M., de Camargo, R. Y., Filho, F. C. & Costa, F. M. (2010). Application execution management on the integrade opportunistic grid middleware, *Journal of Parallel and Distributed Computing* 70(5): 573 – 583.

de Camargo, R. Y., Cerqueira, R. & Kon, F. (2006). Strategies for checkpoint storage on opportunistic grids, *IEEE Distributed Systems Online* 18(6).

de Camargo, R. Y. & Kon, F. (2007). Design and implementation of a middleware for data storage in opportunistic grids, *CCGrid '07: Proceedings of the 7th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA.

de Camargo, R. Y., Kon, F. & Goldman, A. (2005). Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments, *SBAC-PAD'05: The 17th International Symposium on Computer Architecture and High Performance Computing*, Rio de Janeiro, Brazil, pp. 226–233.

de Ribamar Braga Pinheiro Júnior, J. (2008). *Xenia: um sistema de segurança para grades computacionais baseado em cadeias de confianą*, PhD thesis, IME/USP.

de Ribamar Braga Pinheiro Júnior, J., Vidal, A. C. T., Kon, F. & Finger, M. (2006). Trust in large-scale computational grids: An SPKI/SDSI extension for representing opinion, *4th International Workshop on Middleware for Grid Computing - MGC 2006*, ACM/IFIP/USENIX, Melbourne, Australia.

Dong, F. & Akl, S. G. (2006). Scheduling algorithms for grid computing: State of the art and open problems, *Technical Report 2006-504*, School of Computing, Queens University, Kingston, Ontario.

El-Rewini, H., Lewis, T. G. & Ali, H. H. (1994). *Task scheduling in parallel and distributed systems*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Elnozahy, M., Alvisi, L., Wang, Y.-M. & Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys* 34(3): 375–408.

Everitt, B., Landau, S. & Leese, M. (2001). *Cluster Analysis*, 4th edn, MIT Press.

Filho, F. C., Castro, R., Marques, A., Neto, F. S., de Camargo, R. Y. & Kon, F. (2008). A group membership service for large-scale grids, *6th International Workshop on Middleware for Grid Computing*, Leuven, Belgium.

Filho, F. C., Castro, R., Marques, A., Soares-Neto, F., de Camargo, R. Y. & Kon, F. (2009). A robust and scalable group membership service for large-scale grids *(in portuguese)*, *SBRC'2009 Workshop on Grid Computing and Applications*, Recife, Brazil.

Finger, M., Bezerra, G. C. & Conde, D. M. R. (2008). Resource use pattern analysis for opportunistic grids, *6th International Workshop on Middleware for Grid Computing (MGC 2008)*, Leuven, Belgium.

Finger, M., Bezerra, G. C. & Conde, D. M. R. (2010). Resource use pattern analysis for predicting resource availability in opportunistic grids, *Concurrency and Computation: Practice and Experience* 22(3).

Ganesh, A. J., Kermarrec, A.-M. & Massoulie, L. (2003). Peer-to-peer membership management for gossip-based protocols, *IEEE Transactions on Computers* 52(2): 139–149.

Goel, A., Steere, D. C., C, C. P. & Walpole, J. (1999). Adaptive resource management via modular feedback control, *Technical report*, Oregon Graduate Institute, Department of Computer Science and Engineering.

Goldchleger, A., Goldman, A., Hayashida, U. & Kon, F. (2005). The implementation of the bsp parallel computing model on the integrade grid middleware, *3rd International Workshop on Middleware for Grid Computing*, ACM Press, Grenoble, France.

Goldchleger, A., Kon, F., Goldman, A., Finger, M. & Bezerra, G. C. (2004). Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines, *Concurrency and Computation: Practice and Experience* 16(5): 449–459.

Grandinetti, L. (2005). *Grid Computing: The New Frontier of High Performance Computing*, Elsevier.

Hwang, S. & Kesselman, C. (2003). Gridworkflow: A flexible failure handling framework for the grid, *hpdc* 00: 126.

Ibarra, O. H. & Kim, C. E. (1977). Heuristic algorithms for scheduling independent tasks on nonidentical processors, *Journal of the ACM (JACM)* 24(2): 280–289.

Ierusalimschy, R., de Figueiredo, L. H. & Filho, W. C. (1996). Lua - an extensible extension language, *Software: Practice Experience* 26(6): 635–652.

Karablieh, F., Bazzi, R. A. & Hicks, M. (2001). Compiler-assisted heterogeneous checkpointing, *SRDS '01: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, USA, pp. 56–65.

Karonis, N., Toonen, B. & Foster, I. (2003). Mpich-g2: a grid-enabled implementation of the message passing interface, *Journal of Parallel and Distributed Computing (JPDC)* 63(3): 551–563.

Lee, C. & Talia, D. (2003). Grid programming models: Current tools, issues and directions, *Grid Computing*, Wiley Online Library, pp. 555–578.

Litzkow, M., Tannenbaum, T., Basney, J. & Livny, M. (1997). Checkpoint and migration of UNIX processes in the Condor distributed processing system, *Technical Report UW-CS-TR-1346*, University of Wisconsin - Madison Computer Sciences Department.

Liu, Y. (2004). Grid scheduling, *Technical report*, Department of Computer Science, University of Iowa. http://www.cs.uiowa.edu/ yanliu/QE/QEreview.pdf, Acessado em: 22/01/2009.

Lopes, R. F., Silva, F. J. S. & Souza, B. B. (2005). Mag: A mobile agent based computational grid platform, *Proceedings of the 4th International Conference on Grid and Cooperative Computing (GCC 2005)*, Springer LNCS Series, Beijing, China.

Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D. & Freund, R. F. (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, IEEE Computer Society, Washington, DC, USA, p. 30.

Maia, R., Cerqueira, R. & Cosme, R. (2006). Oil: an object request broker in the lua, *5th Tools Session of the Brazilian Simposium on Computer Networks (SBRC2006)*, Curitiba, Brazil.

Malluhi, Q. M. & Johnston, W. E. (1998). Coding for high availability of a distributed-parallel storage system, *IEEE Transactions Parallel Distributed Systems* 9(12): 1237–1252.

Medeiros, R., Cirne, W., Brasileiro, F. V. & Sauvé, J. P. (2003). Faults in grids: why are they so bad and what can be done about it?, *4th IEEE International Workshop on Grid Computing (GRID 2003)*, Phoenix, USA, pp. 18–24.

MPI (1997). *MPI-2: extensions to the Message-Passing Interface*.
URL: *http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*

MPI (2009). *MPI: A Message-Passing Interface Standard Version 2.2*.
URL: *http://www.mpi-forum.org/docs/mpi22-report/mpi22-report.htm*

OMG (2011). *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1.1*, Object Management Group (OMG).
URL: *http://www.omg.org/spec/CORBA/3.1.1/*

Pham, V. A. & Karmouch, A. (1998). Mobile software agents: An overview, *Communications Magazine* 7(36): 26–37.

Pinheiro, V. G., Goldman, A. & Kon, F. (2011). Adaptive fault tolerance mechanisms for opportunistic environments: a mobile agent approach, *Concurrency and Computation: Practice and Experience* .

Plank, J. S., Li, K. & Puening, M. A. (1998). Diskless checkpointing, *IEEE Transactions on Parallel and Distributed Systems* 9(10): 972–986.

Rabin, M. O. (1989). Efficient dispersal of information for security, load balancing, and fault tolerance, *Journal of the ACM* 36(2): 335–348.

Ripeanu, M. & Foster, I. (2002). A decentralized, adaptive replica location mechanism, *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society, Washington, DC, USA.

Rodrigues, R. & Liskov, B. (2005). High availability in DHTs: Erasure coding vs. replication, *IPTPS '05: Revised Selected Papers from the Fourth International Workshop on Peer-to-Peer Systems*, Springer-Verlag, London, UK, pp. 226–239.

Rowstron, A. I. T. & Druschel, P. (2001). Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems, *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, pp. 329–350.

Schopf, J. M. (2004). Ten actions when grid scheduling: the user as a grid scheduler, *Grid resource management: state of the art and future trends* pp. 15–23.

Sobe, P. (2003). Stable checkpointing in distributed systems without shared disks, *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, Washington, DC, USA, p. 214.2.

Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C. & Walpole, J. (1999). A feedback-driven proportion allocator for real-rate scheduling, *OSDI âĂŹ99: Proceedings of the Third Symposium on Operating Systems Design and Implementation*, USENIX Association, New Orleans, LA, USA, pp. 145–158.

Strumpen, V. & Ramkumar, B. (1996). Portable checkpointing and recovery in heterogeneous environments, *Technical Report UI-ECE TR-96.6.1*, University of Iowa.

Subramani, V., Kettimuthu, R., Srinivasan, S. & Sadayappan, P. (2002). Distributed job scheduling on computational grids using multiple simultaneous requests, *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society, Washington, DC, USA, p. 359.

Theodoridis, S. & Koutroumba, K. (2003). *Pattern Recognition*, Elsevier Academic Press.

Valiant, L. G. (1990). A bridging model for parallel computation, *Communications of the ACM* 33(8): 103 – 111.

van Renesse, R., Minsky, Y. & Hayden, M. (1998). A gossip-style failure detection service, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 1998)*, Lake District, England.

Weatherspoon, H. & Kubiatowicz, J. (2002). Erasure coding vs. replication: A quantitative comparison, *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, Springer-Verlag, London, UK, pp. 328–338.

Wilkinson, B. (2009). *Grid Computing: Techniques and Applications*, Chapman & Hall/CRC.

Zhu, Y. (2003). A survey on grid scheduling systems, *Technical report*, Computer Science Department, University of Science and Technology, Hong Kong.