J Braz Comput Soc (2010) 16: 177–190
DOI 10.1007/s13173-010-0016-0

ORIGINAL PAPER

Reliable management of checkpointing and application data in opportunistic grids

Raphael Y. de Camargo · Fernando Castor · Fabio Kon

Received: 22 January 2010 / Accepted: 17 June 2010 / Published online: 28 July 2010

Abstract Opportunistic computational grids use idle processor cycles from shared machines to enable the execution of long-running parallel applications. Besides computational power, these applications may also consume and generate large amounts of data, requiring an efficient data storage and management infrastructure. In this article, we present an integrated middleware infrastructure that enables the use of not only idle processor cycles, but also unused disk space of shared machines. Our middleware enables the reliable distributed storage of application data in the shared machines in a redundant and fault-tolerant way. A checkpointing-based mechanism monitors the execution of parallel applications, saves periodical checkpoints in the shared machines, and in case of node failures, supports the application migration across heterogeneous grid nodes. We evaluate the feasibility of our middleware using experiments and simulations. Our evaluation shows that the proposed middleware promotes important improvements in grid data management reliability while imposing a low performance overhead.

Keywords Grid computing · Distributed data storage · Opportunistic grid · Grid middleware

1 Introduction

Opportunistic computational grids [15, 16, 20, 27] use idle resources from shared commodity machines to execute applications that need large amounts of computational power. The classical scenarios for opportunistic grids are universities, research labs, and organizations that typically have hundreds or thousands of computers that remain idle for most of the time. The objective is to provide users with large amounts of computational power and storage space with a very low cost, since there is no need for acquiring new hardware and for extra physical infrastructure, including electricity, air-conditioning, and physical space.

The concept of opportunistic grid computing can also be very useful in dedicated grid infrastructures, such as Grid'5000 (http://www.grid5000.fr), which comprise thousands of machines connected by dedicated high-speed networks. To use computational resources from this grid, it is necessary to make a reservation in advance and the user has total access to them. However, for applications that require hundreds of nodes for several weeks, it is unfeasible to reserve those resources for such a long time. Grid'5000 includes an opportunistic mode where a user can leverage its resources while no other user places a reservation on them. This mode is rarely used in practice though since the processes running on those machines are killed, and need to be restart from the beginning whenever the machines are

This research was supported by CNPq/Brazil, grants #481147/2007-1 and #550895/2007-8.

R.Y. de Camargo (✉)
Center for Mathematics, Computation and Cognition,
Federal University of ABC (UFABC), R. Catequese, 242,
Santo André/SP, 09090-400, Brazil
e-mail: raphael.camargo@ufabc.edu.br

F. Castor
Informatics Center, Federal University of Pernambuco (UFPE),
Recife/PE, Brazil
e-mail: castor@cin.ufpe.br

F. Kon
Department of Computer Science, University of São Paulo (USP),
São Paulo/SP, Brazil
e-mail: kon@ime.usp.br

requested. If one could deploy an opportunistic grid middleware that uses those thousands machines when they are idle, one would have a much better resource utilization level. In the case of Grid'5000, if we consider that machines are idle for about 50% of the time, we would double the capacity of this multimillion euro infrastructure. Moreover, each machine has in the order of a hundred Gigabytes of unused disk space, which combined can easily reach about 100 Terabytes of distributed storage space, connected by high-speed networks.

In current opportunistic grid systems, resource sharing is limited to idle processor cycles and main memory [15, 16, 20, 27]. A middleware system that enables the sharing of both processor cycles and unused disk space would improve resource utilization in the grid machines and increase the amount of available resources. The most important difference of opportunistic grids, when compared to dedicated ones, is that machines will very often fail, become inaccessible, or change from idle to occupied unexpectedly. This may compromise the middleware infrastructure, the data stored in the nodes that became unavailable, and the execution of grid applications. Consequently, an opportunistic grid middleware must be fault-tolerant regarding its infrastructure, stored data, and application execution.

In this article, we present and evaluate an integrated middleware system, based on the InteGrade [20] grid system and OppStore [10] distributed storage system, that enables the usage of both idle processor cycles and unused disk space of shared machines. The middleware enables reliable application execution using a checkpointing-based fault tolerance mechanism for parallel applications, which generates periodic checkpoints that contain the application state in the moment when they were generated and stores the checkpoints in the grid shared machines in a redundant and fault-tolerant way. The mechanism then uses the stored checkpoints to recover the application state after a failure, allowing application recovery in heterogeneous grid nodes.

To allow storage and management of application input, output, and checkpointing data, we organize the grid machines in clusters, connected by a self-organizing and fault-tolerant peer-to-peer network. Our middleware stores the data in a distributed fashion and the data is codified into redundant fragments, enabling the reconstruction of the original file using only a subset of those fragments. Stored data can be accessed from any machine in the grid and fragments can be downloaded from the machines with the fastest connections. To deal with resource heterogeneity, we extended the Pastry [37] peer-to-peer routing substrate to support virtual ids, which enable the selection of clusters containing machines with higher availability for data storage.

Our experiments and simulations show that our approach is viable. In the evaluated scenarios, our middleware enabled the immediate retrieval of stored files in over 99% of the requests, improved retrieval performance by transferring data in parallel from several distributed machines, and provided fault-tolerance in the presence of machine departures or crashes, by reconstructing the lost fragments. The checkpointing mechanism has low overhead and checkpoints are stored in a reliable way, enabling the execution of coupled parallel applications even in the presence of node failures.

The main contributions of this work are:

- development of an integrated middleware infrastructure that enables the use of both idle processor cycles and unused disk space of shared machines in opportunistic grids;
- usage of a new approach to store checkpoint and grid application data in a set of non-dedicated, heterogeneous machines that is highly fault-tolerant and available and, at the same time, supports fast retrieval of checkpoints when faults occur; and
- evaluation through experiments and simulations of the feasibility of using the idle disk space of grid machines for storage of checkpointing and application data.

2 Related work

Our work encompasses several research areas, but we focus here on the literature related to the original contributions of this article, i.e., management of checkpoints of parallel applications, distributed data storage, and grid data management.

In a previous work, we presented OppStore [10], a middleware for distributed storage of data that could be adapted to work with existing opportunistic grid systems. In this article, we expand that work by integrating OppStore with the InteGrade application execution protocol and checkpointing mechanism, resulting in a middleware that enables the use of both idle processor cycles and unused disk space of shared machines in a opportunistic grid. In addition, we discuss the efficient storage of checkpointing data, management of grid application input and output data, and the reconstruction of fragments lost due to node departures.

2.1 Distributed data storage and management

Our middleware has some similarities with other distributed storage systems built over P2P networks. PAST [36], CFS [17], and pStore [2] are peer-to-peer distributed storage systems built over a DHT (Distributed Hash Table) infrastructure, such as Pastry [37]. To improve availability, PAST stores multiple replicas of the files and CFS and pStore break the files into fragments and store several replicas of those fragments. The main difference to our system is that PAST, CFS, and pStore organize all machines at a single level and store data directly in the machines that perform message routing. Consequently, each node arrival and

departure requires that large amounts of data be transfered to compensate for changes in machine organization, as described by Blake and Rodrigues [4]. PeerStore [26], like our work, decouples fragment location from its identifier, storing the file index and fragments separately, reducing the maintenance due to data misplacement caused by node joinings and departures. Notwithstanding, it recovers lost fragments caused by departures in a lazy way, compromising the immediate recovery of stored files.

FreeLoader [42] uses free storage space from machines of a single cluster and unused I/O bandwidth to store scientific data. They use data striping to divide a file into several fragments to improve performance. Similarly to our work, they target nondedicated resources for data storage. But they only consider static sets of machines from a single cluster, while we deal with dynamic sets of machines distributed across several clusters. Also, they do not consider load-balancing and machine availability when choosing storage sites.

In the area of grid data management, JuxMem [1] implements a data sharing service for grid applications by joining the concepts of peer-to-peer and distributed shared memory. As in our work, they organize grid machines as a federation of clusters organized as a peer-to-peer network and with a node elected as cluster manager in each cluster. Differently from our work, they focus on the development of a writable distributed shared memory for grid applications. This requires maintaining several full replicas of stored data, which incurs large storage and network overheads, especially when operating with nondedicated machines, where the replication level must be higher.

A common technique for data grids is the usage of data replication in conjunction with a replica location system [8, 13, 35]. Some replica management systems [13, 35] use compression schemes, such as Bloom filters [5], allowing replica managers to have complete knowledge about replica locations on the grid. The search mechanism is fast and the system has a high degree of fault-tolerance. Nevertheless, due to global knowledge of replica locations, these systems have limited scalability, with possibly expensive table updates. Also, the servers maintaining the replica locations are usually configured statically. Cai, Chervenak, and Frank [8] built a replica location system using the Chord [39] DHT, to provide self-organization and improved fault-tolerance and scalability for replica location systems. But this system only deals with replica location, while our system also deals with storage in nondedicated repositories, including the selection of appropriate repositories.

2.2 Storage of checkpointing data

Pruyne and Livny [33] performed studies about the usage of multiple checkpoint servers to store checkpoints from parallel applications. They only compare the usage of single and dual checkpoint servers and the servers were dedicated. Plank, Li, and Puening [32] propose the usage of diskless checkpointing. It consists of storing checkpointing data on system volatile memory, removing the overhead of stable storage. The focus of their work was on comparing diskless with disk-based checkpointing and their experiments were performed using only parity information for fault tolerance.

Malluhi and Johnston [29] compare analytically the efficiency of an optimized version of the Information Dispersal Algorithm (IDA) proposed by Rabin [34] and 2D parity coding schemes. Sobe [38] analyzes the use of two different parity techniques to store checkpoints in distributed systems. Differently from our work, they focused exclusively on the storage of data and conducted a purely analytical evaluation, without performing any experiments.

In the area of grid computing, Luckow and Schnor [28] propose an adaptive replication mechanism for the storage and management of checkpoint files in the context of grid computing. Differently from our work, they distribute the replicas of checkpoint files in remote sites. This approach promotes an improvement in reliability but, in the context of opportunistic grids, is prohibitively expensive in terms of both disk and bandwidth consumption.

3 The middleware architecture

Our middleware organizes the machines in clusters, where each cluster contains physically close machines. We use InteGrade [20], a CORBA-based object-oriented opportunistic grid system, to perform the management of application execution. For the storage of checkpointing and application data, we use modules from OppStore [10], a distributed storage system that allows the reliable storage of application data in the unused space of idle machines.

Figure 1 shows the organization of the grid machines in grid clusters and the middleware modules in the cluster machines. Each cluster has a machine designated as the *cluster manager*, usually a machine that is available for most of the time. Application execution is controlled by the Global Resource Manager (GRM), responsible for scheduling and managing the cluster resources, the Execution Manager (EM), responsible for managing application execution and maintaining checkpointing information, and the Application Repository (AR), which maintains information about grid application files, such as their executable, input, and output files. Data storage is managed by the Cluster Data Repository Manager (CDRM), which is responsible for supervising the machines within the same cluster. The *resource provider* machines share their idle resources with the grid. The Local Resource Manager (LRM) module manages the machine shared resources and local processes associated

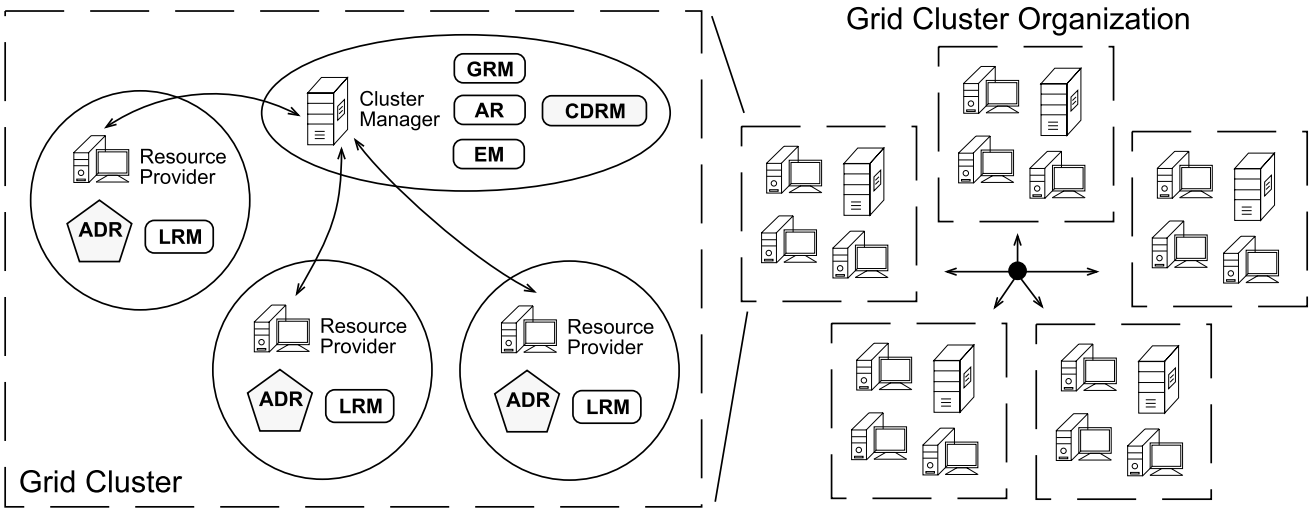

Fig. 1 Organization of the shared machines in clusters and distribution of the main middleware modules

with grid applications and the Autonomous Data Repository (ADR) module manages the shared unused disk space on the machine. The modules executed in the resource provider machines are written in C++ and use a lightweight CORBA ORB, called OiL (http://oil.luaforge.net), which results in a low execution and memory overhead of the middleware modules executing on these machines.

The integrated middleware, based on InteGrade and OppStore, can be downloaded as open-source software from http://www.integrade.org.br.

4 Reliable distributed storage

The modules responsible for the management of the reliable distributed storage are the Autonomous Data Repository (ADR), which runs on each resource provider machine and controls the storage and retrieval of stored data in the machine, and the Cluster Data Repository Manager (CDRM), which supervises all the machines sharing disk space on its cluster.

In this section, we describe the organization of the CDRM and ADR modules, the procedures for file storage and retrieval, and the protocols for guaranteeing data availability in the presence of machine departures.

4.1 CDRM organization

We organized the CDRMs in a *peer-to-peer* network structured as a distributed hash table (DHT) that leverages Pastry [37] as its substrate. Pastry assigns to each node (a CDRM in our case) a random identifier, called Pastry id, from a range of valid identifiers, called Pastry id space. Users can now route messages, each containing a message id, that reach the node with node id closest to the message id. Routing usually requires several hops, where on each hop the message reaches a node with an id increasingly closer to message id, until it reaches the target node. The message is routed through $O(\log n)$ nodes, where n represents the number of machines in the system.

A problem with Pastry is that the heterogeneity of nodes is not considered. To deal with this limitation, we assign to each node an additional identifier, called virtual id [9], using the same range of valid identifiers from Pastry, which we now call virtual id space. In our system, each CDRM receives a virtual id and becomes responsible for a virtual id range proportional to the capacity of the cluster. We defined the capacity of the cluster as the sum of the capacities of its machines, which is proportional to their mean availability and free disk space. The capacity of the clusters can change due to machine departures or variations in their free space or availabilities. If the change in the cluster capacity is above a threshold, the system adjusts the virtual id of the CDRM to reflect the new capacity.

Routing using virtual ids is performed using the Pastry routing infrastructure, with the difference that each CDRM keeps an extra table containing the virtual ids of its logical neighbors, which is used to guide the Pastry routing process. Most of the work of maintaining the peer-to-peer infrastructure is also performed by Pastry, with the additional task of updating the table containing the node logical neighbors. We use the FreePastry implementation of Pastry, which allowed us to implement the virtual ids with small effort.

We use the DHT to locate the clusters where the middleware will store the file fragments and to retrieve the fragments and reconstruct those files later. The advantage of using Pastry is that it provides a routing algorithm that is fault-tolerant and self-organizing, allowing the system to deal cor-

rectly with cluster departures and CDRM failures. In addition, by implementing an extension to Pastry, called virtual ids, we could also deal correctly with resource heterogeneity.

4.2 Data storage and retrieval

Clients access the distributed storage system through the *access broker* library, which is responsible for contacting the CDRM and ADR modules to perform file storage and retrieval operations. Several types of data can be stored in a grid system, with each type having different requirements. In this work, we do not make any assumptions about the internal organization of the data to be stored. Instead, we focus on the requirements of applications, in terms of the expected lifetime of the data they store. Our system lets a client application choose one of two storage modes: perennial and ephemeral.

The *perennial mode* is used for data with long lifetimes. In this mode, the broker encodes the files into redundant fragments, using an optimized version [29] of the Information Dispersal Algorithm (IDA) developed by Rabin [34]. This coding generates n fragments, from which any $k \leq n$ are sufficient to reconstruct the original file and the total fragment size is n/k times the original file size. Using this encoding, one can tolerate $n - k$ failures with an overhead of only $(n - k)/k$ times the original file size. Weatherspoon and Kubiatowicz [43] showed that, using the same amount of storage space, IDA provides higher reliability than replication, since it can tolerate a higher number of concurrent periods of node unavailability.

The broker codes the file into fragments and evaluates their secure hash, which we call their *id*, as shown in Fig. 2. To select the machines that will store the fragments, the CDRM routes a message for each fragment, which is delivered to the CDRM responsible for the fragment *id* in the virtual id space (step 2), guaranteeing that selection of clusters with higher capacity will be more likely. Each CDRM then selects one ADR from its cluster, with a chance proportional to the ADR capacity, and returns the address of the ADR (step 3). After receiving the addresses, the broker transfers the fragment contents directly to the ADRs (step 4). Finally, at step (5), the broker constructs a *File Fragment Index* (*FFI*) containing the ADR address of each fragment and some extra information, hashes the FFI contents to generate a unique file *id*, and requests the FFI storage in the CDRM responsible for the file *id*, with additional copies stored on its two logical neighbors for fault-tolerance.

To download a file, the broker queries its cluster CDRM for the file FFI. The CDRM routes the request to the CDRM responsible for the file *id*, which returns the FFI. The broker then downloads the file fragments directly from the ADRs, checks the fragments integrity and reconstructs the file.

We should emphasize that file contents are not routed through the overlay network. They are transferred directly

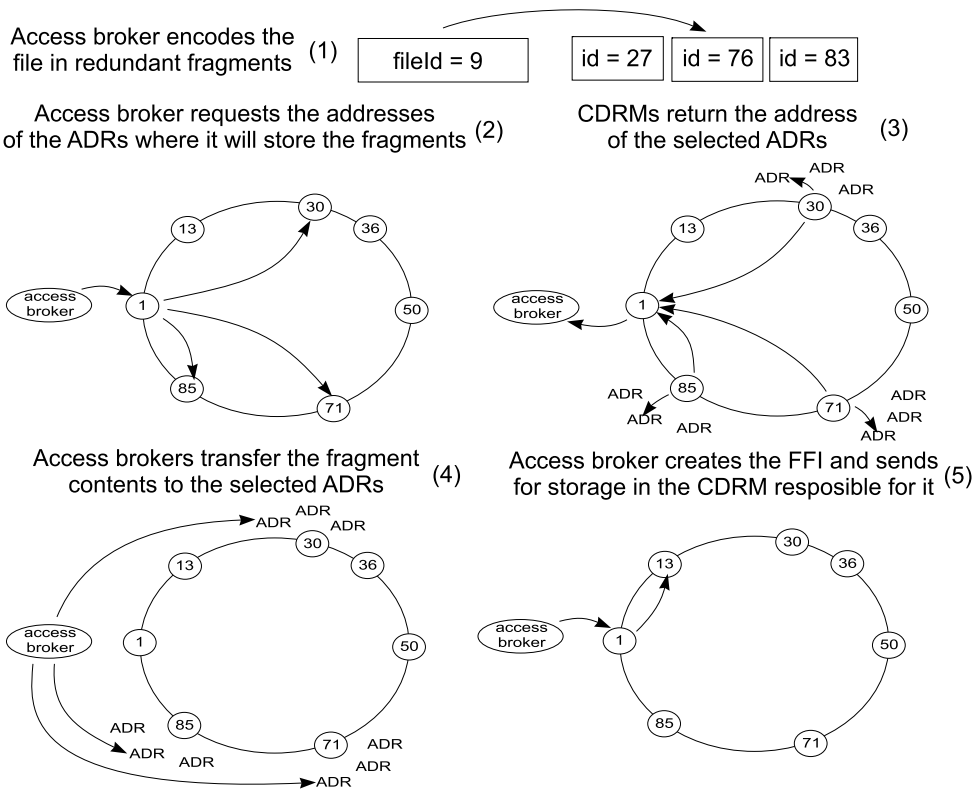

Fig. 2 Protocol for storage of a file containing application data using the perennial mode

from the broker to the ADRs and vice versa. Also, placing fragment storage locations in the FFI provides an important advantage: the fragment locations are not tied to the cluster responsible for their *id*s. In other words, when the *id* range for which a CDRM is responsible changes, there is no need to move the fragments across clusters. The FFIs are still retrieved using the file *id*s, and consequently, would need to be moved, but their size is small compared to file contents.

The *ephemeral mode* is used for data that requires high bandwidth and only needs to be available for a few hours. This class of storage is used to store checkpoints generated by our checkpointing mechanism, as we will see in Sect. 5.3. Another usage is for storing temporary data, such as in workflow applications, where data output by one application stage is used by a later application stage running in the same cluster. In this storage mode, two replicas of the file are stored in machines from the same cluster where the storage request was issued. Another option would be to use IDA to distribute the file fragments in the cluster but, as we will see in Sect. 5.3, in this case it is advantageous to use replication.

4.3 Data management

Node joins and departures are frequent in opportunistic grid environments. Although the use of erasure coding improves data availability significantly [43], when a machine leaves the grid, fragments are lost and need to be reconstructed. Moreover, a machine owner may need the disk space that it shared with the grid, requiring our middleware to immediately remove the fragments from the machine. Cluster departures are also possible. To prevent data losses and increase data availability in the presence of node departures, we developed protocols that enable the reconstruction of lost fragments.

Our system must determine which files should be monitored and reconstructed. The system uses the leasing pattern [24], providing a lease to each stored file with a duration specified by the client. These leases can be renewed by clients at any time. During the leasing period, the fragment reconstruction protocol is executed whenever necessary. When the leasing period ends, the file and fragments are marked as expired, meaning that they are not reconstructed and can be removed by the ADRs.

Fragment reconstruction protocol

We designed a reconstruction protocol for lost fragments to guarantee that data is not lost when nodes depart from the system. The reconstruction protocol works as follows:

1. The CDRM from the cluster where a machine departed notifies the CDRMs containing the FFIs of each fragment that was stored in the departed machine.
2. For each notification a CDRM receives, it verifies whether the *reconstruction threshold* was reached. The reconstruction threshold is the minimum number of fragments that need to be available for a given file. If the threshold was reached, the CDRM downloads the fragments necessary to reconstruct the original file directly from the ADRs.
3. The CDRM reconstructs the lost fragments from the file and, for each generated fragment, routes a message requesting an ADR address to store the fragment.
4. The CDRM transfers the generated fragments to their storage sites and updates the FFI of the files with recovered fragments with the new storage sites.

This protocol is fault-tolerant, since failures in the reconstruction of some fragments in steps 2 and 3 results in an FFI with a smaller number of stored fragments, but still above the reconstruction threshold. Also, if part of the information in the FFI becomes stale during the reconstruction protocol, such as when other fragments are lost, the algorithm still works, since it will reconstruct the fragments that were missing when the protocol started. Finally, the protocol should also work in the case of burst-like workloads, since even if most of the resources of a cluster are suddenly allocated to a user, the fragments stored in other clusters in the grid could still be recovered.

It is difficult to know whether a node departure is temporary or permanent. We use heartbeats as a way to detect node departures and, consequently, false departure detections can happen in the case of transient network failures. To minimize the impact of transient departures and false node departure detection, we delay the beginning of the reconstruction until reaching the reconstruction threshold, in the hopes that some of the nodes left the grid only temporarily or that the departure was a false positive. The reconstruction threshold value is set at the midpoint between the number of fragments required to reconstruct the file and the total number of fragments. For instance, if a file is encoded in 9 fragments, from which 3 are sufficient to recover the file, the reconstruction threshold is set to 6.

Another important traffic generator event occurs when machines join or leave the system. In this case, it is necessary to correct data misplacement due to the reorganization of the *id* space [4, 26]. Since our system decouples the fragment storage location from its *id*, large data transfers are not required during machine joins and departures.

Local fragment copy

The fragment reconstruction protocol presented above can reconstruct lost fragments in all scenarios, but incurs in a large amount of communication. However, it is possible to avoid starting the fragment reconstruction process in the majority of cases by keeping, for each fragment stored in cluster, an extra copy of that fragment in another machine from

the same cluster. When a machine departs, the CDRM can immediately regenerate each of the lost fragments using the copies stored in the cluster. In other words, machine departures are treated locally within the cluster.

In this approach, we have to double the required storage space, but fragments are lost only when the two machines holding the copies of a fragment fail or leave the grid almost simultaneously. If we combine the local fragment copy with a reconstruction threshold, we eliminate the need for the reconstruction protocol in the vast majority of cases. Moreover, keeping local fragment copies puts no extra burden on the intercluster connections, which normally have a more limited amount of bandwidth than local networks. Consequently, if storage space is not scarce, we can use this approach to solve the data maintenance problem.

The usage of local network bandwidth for reconstructing lost fragments using the local fragment copy is also lower. In the reconstruction protocol without the local fragment copy, every time the fragments are transferred in the intercluster network, they are also transferred in the local networks of two different clusters. More details can be found in [12].

5 Reliable execution of parallel applications

Opportunistic grid middleware systems should allow the execution of parallel applications in the presence of frequent periods of machine unavailability. InteGrade has a checkpointing mechanism that generates and stores periodical checkpoints containing the application state. The Execution Manager (EM) module monitors the application execution and, when it detects failures during the execution, it restarts the application from the last stored checkpoint.

But 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 in a reliable way. We extended InteGrade to use the unused disk space of the shared grid machines to store this data in a reliable and fault-tolerant way.

In this section, we describe how the storage of checkpointing and application data enables the reliable execution of sequential and parallel applications. We start the section describing how we implemented the management of application data in InteGrade. Next, we describe the checkpoint generation process in InteGrade and finish the section comparing the storage methods we use to store checkpointing data.

5.1 Management of application data

The execution of an application on InteGrade [20] starts with the user registering its executable in the Application Repository (AR). The user then requests the execution of the application, providing the identifier of the stored binary file and a list of input files, located in the user submission machine, that will be used by the grid application. The user requests the execution to the Global Resource Manager (GRM) of its cluster, which selects the machines (LRMs) that will execute the application. The selected LRMs then download the binary file from the AR and the input files from the machine where the user submitted the execution request. When the execution finishes, the LRMs send the output files directly to the user machine. All the execution process is monitored by the Execution Manager.

The main limitation of this approach is that the user submission machine must stay online during all the execution process, since input and output files are transferred directly to/from it. Moreover, the execution may not start immediately after the request and the execution can last for several days.

To solve this problem, we modified InteGrade to store application input and output data using the perennial mode. When the user submits an application for execution, it can either use input data already stored in the distributed nodes or provide new input data, which will be uploaded for storage. When the application finishes its execution, its results are stored in the distributed nodes, and the user can retrieve the results immediately or later. We use the Application Repository (AR) module to maintain the mapping of the executable, input, and output files to file *id*s used to locate the files stored using the perennial mode.

5.2 Checkpointing mechanism

To provide fault-tolerance, InteGrade has a checkpoint-based rollback recovery mechanism [11, 19], which saves the application state into checkpoints and, in case of node failure, reinitializes the application from the state contained in the last saved checkpoint. The mechanism works for sequential, bag-of-tasks, and BSP coupled parallel applications. The BSP (Bulk Synchronous Parallel) model [41] divides the computation into a sequence of parallel supersteps, where each superstep is composed of a computation and a communication phase, followed by a barrier of synchronization. Since communication is always followed by a synchronization barrier, it is possible to avoid the well-known problems of distributed checkpointing [25] by only creating checkpoints after the synchronization step.

InteGrade uses application-level checkpointing [7, 23, 40], which consists of instrumenting application source code to save its state periodically. It contrasts with traditional system-level checkpointing, where the data is saved directly from the process virtual memory space by a separate process or thread [31]. Since in application-level checkpointing one manipulates application source code, semantic information regarding the type of data being saved is available both when saving and recovering application

data. This semantic information allows the data saved by a process on a hardware architecture to be recovered by a process executing on another hardware architecture and/or on a different operating system [23, 40]. This is an important advantage for a Grid composed of heterogeneous machines since it enables better resource utilization. The main drawback of application-level checkpointing is that manually inserting code to save and recover application state is a tedious and error prone process. To resolve this problem, InteGrade provides a precompiler that automatically inserts the required code. It works for applications written in C or in a subset of C++ and is based on OpenC++ [14], an open source tool for compile time reflective computing. Although applications in other languages are not supported by the precompiler, checkpoints for these applications could be obtained using existing system-level checkpoint tools, such as libckpt [31] and CryoPID [3], with the drawback that portability would be lost.

In the case of coupled parallel applications, such as BSP applications, it is necessary to guarantee that the generated checkpoints are consistent. The InteGrade implementation of the BSP model, called BSPlib [21], contains extra functions that can be used to control the generation of checkpoints. The precompiler also analyzes the BSP calls and modifies them to allow the generation of consistent checkpoints.

The Execution Manager (EM) monitors application execution and coordinates the recovery of failed applications. It maintains a list of active processes executing on each node and a list of the processes from each parallel application. When the GRM detects that a resource provider machine is unreachable, it notifies the EM, which reschedules the applications that had processes executing on the failed machines for execution in another machine. The application is then restarted, using the last generated checkpoint and its input files.

5.3 Checkpoint storage

We modified InteGrade to use the ephemeral mode of OppStore to store most of the generated checkpoints, 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 5 generated checkpoints. With this strategy, we can obtain low overhead and high availability at the same time.

We implemented the management of which checkpoint belongs to which application in the Execution Manager and used the OppStore modules exclusively for checkpoint storage and retrieval. There are two strategies that we can use for storing the checkpoints in the cluster, which are data replication and IDA.

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.

In our scenario, transferring checkpointing data twice would generate too much local network traffic, possibly compromising the execution of the parallel application, so the approach we adopted was to store a copy of the checkpoint locally and another remotely. 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 provides recovery as long as one of the two nodes containing a checkpoint replica is available.

The other strategy is to use IDA, which we described in Sect. 4.2. 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. We compare the checkpointing overhead in the execution time of a parallel application when using IDA and replication to store the checkpoints in Sect. 6.5.

6 Middleware evaluation

We evaluated the feasibility of using our middleware to store application data in the unused disk space of shared machines and the overhead to periodically generate and store checkpoints. We used experiments and simulations to evaluate both the performance in practical deployment environments and the large-scale properties of our system.

6.1 Data availability

We used simulations to evaluate the availability of data stored in the middleware using the perennial mode, simulating file retrieval operations in realistic scenarios. In the simulations, we instantiate several CDRMs in a single Java Virtual Machine and execute the protocols using our middleware implementation. We only simulate the ADRs, the access broker, and fragment transfers, for which we implemented a simulator in Java.

We simulated an opportunistic grid composed of 30 clusters of desktop machines. The number of machines in each cluster was randomly chosen among 10, 20, 50, 100, and 200. To evaluate file availability, we used data from several machine utilization measurements [6, 18, 30] to define three

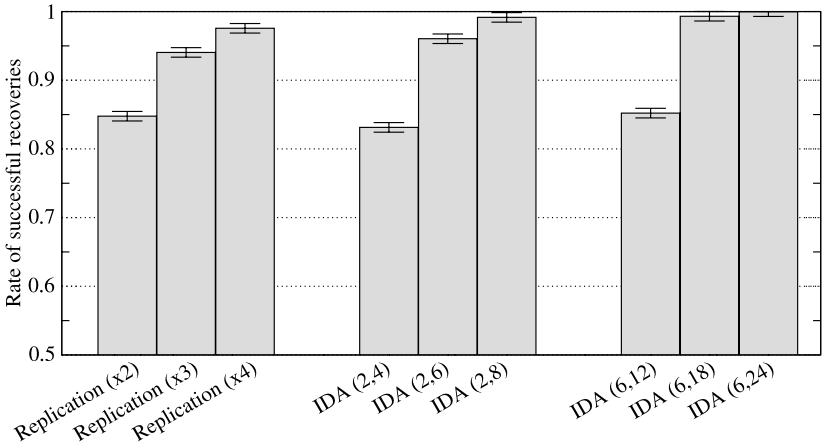

Fig. 3 Rate of successful file retrievals when using IDA and replication to store the files. We used different redundancy levels and number of fragments

different usage patterns, which were randomly assigned to each cluster. In the first pattern, cluster resources are idle, on the average, 60% of the time during the day and 80% of the time during the night and weekends. The second pattern has idle times of 25% and 40%, and the third 40% and 70%, respectively. We consider that clusters are uniformly distributed across 24 time zones.

We consider that machines from a single cluster have similar properties. The simulator randomly assigns one of the usage patterns for all the machines of the cluster. For half of the clusters, each machine receives an amount of free disk space randomly chosen between 1 and 10 GB, and for the other half, they receive values between 20 GB and 50 GB.

We simulated the storage of 1,000 files and evaluated the rate of successful file retrieval using the machine usage patterns described above, considering the scenario where we can recover only in the idle periods of the machines. We stored dummy fragments of a few bytes, which are used only to test their availability during the retrieval simulation. We repeated the simulation 12 times, each with a different random grid configuration, which resulted in a small standard deviation. We considered that during the simulation time there are no ADR and CDRM departures. We analyze the scenario with node departures in Sect. 6.2.

We simulated file retrieval for a period of 1 month. For each retrieval request, machines containing fragments from the file return a dummy fragment, if the machine is idle, or an error, if the machine is unavailable. The state of a machine during a retrieval trial is defined using a Bernoulli distribution, where the chance of the machine being in the idle state depends on the request time (day of the week and hour) and the machine availability pattern, for example, 40% during the weekdays in the third pattern. If the broker retrieves all the fragments necessary to reconstruct a file, we consider the retrieval as successful.

We used several data coding schemes IDA(k, n), where k is the number of fragments required to recover the original file and n is the number of generated fragments, and compared with the usage of fragment replication. We used the schemes IDA(2.4), IDA(2, 6), IDA(2, 8), IDA(6, 12), IDA(6, 18), IDA(6, 24), and replication with 2, 3, and 4 replicas. The objective was to compare the impact of the number of fragments on data availability.

The vertical bars in Fig. 3 show the mean rate of successful file retrievals over 12 simulation runs and the error bars represent the standard deviations. As we can see, the probability that a requested file is not available at the moment a request is made is small using reasonable redundancy levels. For instance, we achieved a successful retrieval rate of 99.3% when using IDA(6, 18), 96.0% when using IDA(2, 6) and 94.0% when storing 3 replicas of the files. The standard deviation was small, due to the large number of files and long simulation period. Besides increasing data availability, using IDA improves retrieval performance, since the access broker can select the fastest servers to download the files.

But even when a file cannot be recovered immediately, the system can wait for the files to become available, either keeping the resources reserved for the application or rescheduling the application for later execution, allowing another application to execute. Since it is not possible to predict when this file will be available, a good choice here would be to reschedule when other applications are waiting in the execution queue or to keep waiting when the execution queue is empty.

6.2 Data availability with node departures

In this simulation, we evaluate the availability of files stored in our middleware in the presence of machine departures. We stored 3,000 files in a grid configured in the same way as Sect. 6.1 and simulated a one-month period with a machine departure rate of 10% per day, running the fragment reconstruction protocols. At the end of this period, we requested the recovery of all stored files and measured the percentage of the files that could be recovered.

We consider that when a machine departs, all the fragments in the ADR are lost. To keep the number of machines

Table 1 Mean data availability using the fragment reconstruction protocol with several values for the reconstruction threshold (t) and the local fragment copy (fc) protocol

IDA	$t = 9$	$t = 12$	$t = 15$	fc
$k = 6, n = 12$	0.45133	–	–	0.84733
$k = 6, n = 18$	0.73833	0.87500	0.95867	1.00000
$k = 6, n = 24$	0.88100	0.92933	0.99733	1.00000

stable, when a machine leaves the grid, another one joins the grid immediately in a random cluster. Finally, we consider that the network is reliable and that messages are always delivered.

Table 1 shows the mean availability for several IDA configurations and threshold values and for the local fragment copy. As we increase the threshold, the mean availability of the stored files also increases, since the fragment recovery protocol is executed more often. However, increasing the reconstruction threshold increases the bandwidth to reconstruct fragments, as we show in the next section.

The availability when using the local fragment copy (Sect. 4.3) is always higher, since lost fragments are immediately reconstructed from their local copies. Actually, when using the local fragment copy, the availability was slightly higher than the results from the experiment with no node departures, since there are two copies of each fragment per cluster.

6.3 Network usage for fragment maintenance

In this simulation, we measure the network traffic generated by the maintenance of files stored with the perennial mode under different ADR departure rates. We stored 3,000 files of different sizes in a simulated opportunistic grid, described in Sect. 6.1, with a total size of about 65 GB. We simulated a period of 1 month, where ADRs depart or lose their stored fragments at different rates per day.

Figure 4 shows the bandwidth used to store the files and recover the lost fragments for several ADR departure rates. The figure emphasizes inter-cluster traffic, since the bandwidth available within local area networks is usually much higher and, therefore, less critical, than the bandwidth of wide-area networks. We have employed IDA(6, 12) with threshold 9, IDA(6, 24) with threshold 9, IDA(6, 24) with threshold 15, and IDA(6, 12) with the local fragment copy strategy.

When not using local fragment copy, the required bandwidth increases linearly with the departure rate. But even when considering a departure rate of 1.25% of the nodes per day (leftmost points), the used bandwidth is about 130 GB to store the files and 150 GB to maintain those files, when using IDA(6, 24) with threshold 9. Consequently, in a single month, an amount of data comparable in size to all of the stored data is transferred between clusters. This amounts to a traffic of approximately 58 KB per second. Also, for higher departure rates, we see that increasing the replication level lowers the bandwidth necessary to maintain the data, for example, when we increase the number of fragments to 24. Reducing the reconstruction threshold also lowers the used bandwidth. But as we saw in the previous experiment, file availability decreases when we decrease the reconstruction threshold.

When using the local fragment copy, for all ADR departure rates, the intercluster bandwidth is constant and lower than the other strategies. This occurs because this bandwidth is used only for the transfer of fragments during file storage.

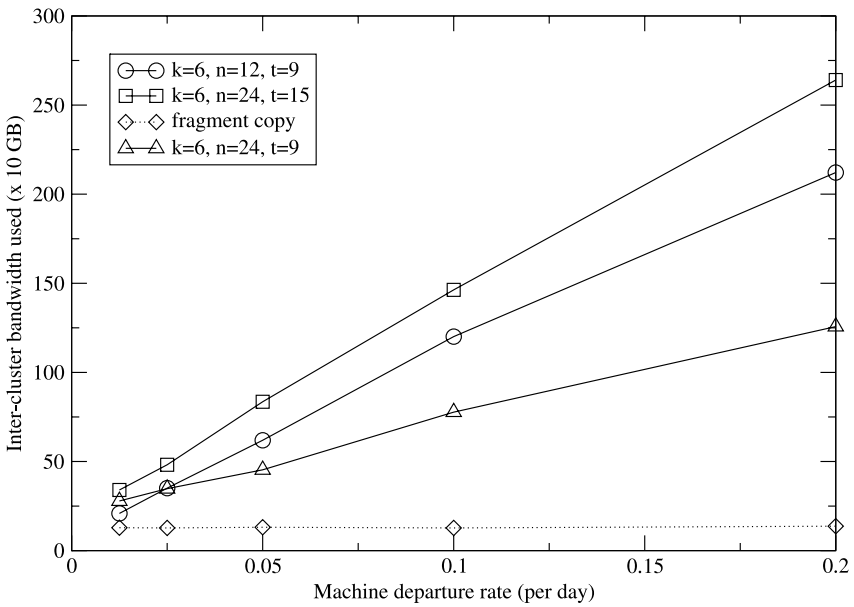

Fig. 4 Bandwidth for fragment reconstruction when fragments are lost due to node departures

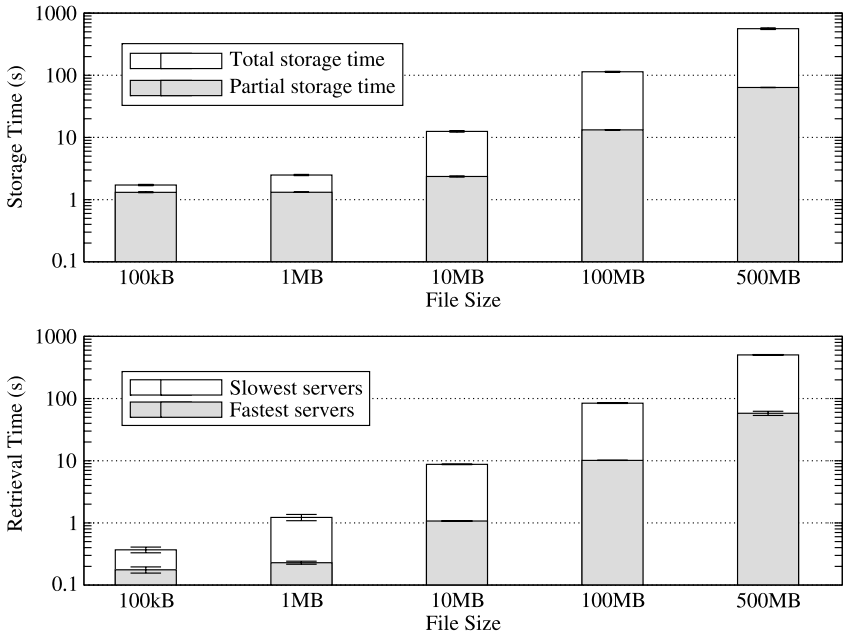

Fig. 5 Results for the time required to store and retrieve files into/from different clusters

Consequently, when considering the intercluster bandwidth used, the local fragment copy strategy is very efficient and the required bandwidth does not increase with the node departure rate.

6.4 Data storage and retrieval

In this experiment, we evaluate the time necessary to store and retrieve files of different sizes using the perennial mode. We used a real grid environment, where we instantiated 5 geographically separated clusters, composed of commodity machines distributed among three Brazilian cities. São Paulo contained three clusters while Goiânia and São Luís contained a single cluster each. They are distant 900 km and 3000 km to São Paulo, respectively. The clusters are connected via the public Internet.

We instantiated the broker in a 2 GHz Athlon64 machine with 4 GB of RAM and stored and retrieved files of sizes 1 MB, 10 MB, 100 MB, and 500 MB. The broker split the file into 5 fragments, from which 2 were sufficient for data recovery. Since our experimental system had only 5 clusters, we forced the fragments to be stored on different clusters. Each experimental session was composed of the storage of one file of each size and its recovery.

In Fig. 5, we show the mean time required to store files and recover stored files and the corresponding standard deviations. For file storage, when a number of fragments sufficient to recover the file is already stored, the broker can already generate a file fragment index (FFI) and store it, while the remaining fragments are stored. When the storage of all fragments finishes, the FFI is updated.

In the figure, the topmost graph shows the time required for both the case where only the fragments sufficient for file recovery are stored and when all fragments are stored. The time to store all the fragments is bounded by the slowest ADRs, specially for large files. In this case, uploading a version of the FFI as soon as a sufficient number of fragments are stored can decrease the storage time by ten. For smaller file sizes, the time for complete and partial storage are similar, since the storage time is bounded by the time to find the ADRs that will store the fragments and to store the FFI.

To recover a file, the broker needs to download only the fragments necessary to reconstruct the file. The choice of the ADRs used to download the fragments is critical when determining the file recovery time. In the bottommost graph, we show the time necessary to recover the fragments when the fastest ADRs and slowest ones are selected. These two lines reflect the extreme cases, with the average situated between them. A good solution to select the fastest ADRs is to start downloading all the fragments simultaneously, to determine the download speeds, and then continue the download only from the fastest ones.

6.5 Storage of checkpoints from parallel applications

To evaluate the viability of using our middleware to provide a reliable execution environment for parallel applications, this experiment measures the impact of the checkpointing mechanism over the application execution time. We compared the time to execute the application without and with checkpointing. In the later case, we store the checkpoint using the ephemeral mode, configured to store the checkpoint using replication and erasure coding.

We selected a matrix multiplication BSP application, developed by Hayashida et al. [22], since it is a highly coupled and long-running parallel application and produces

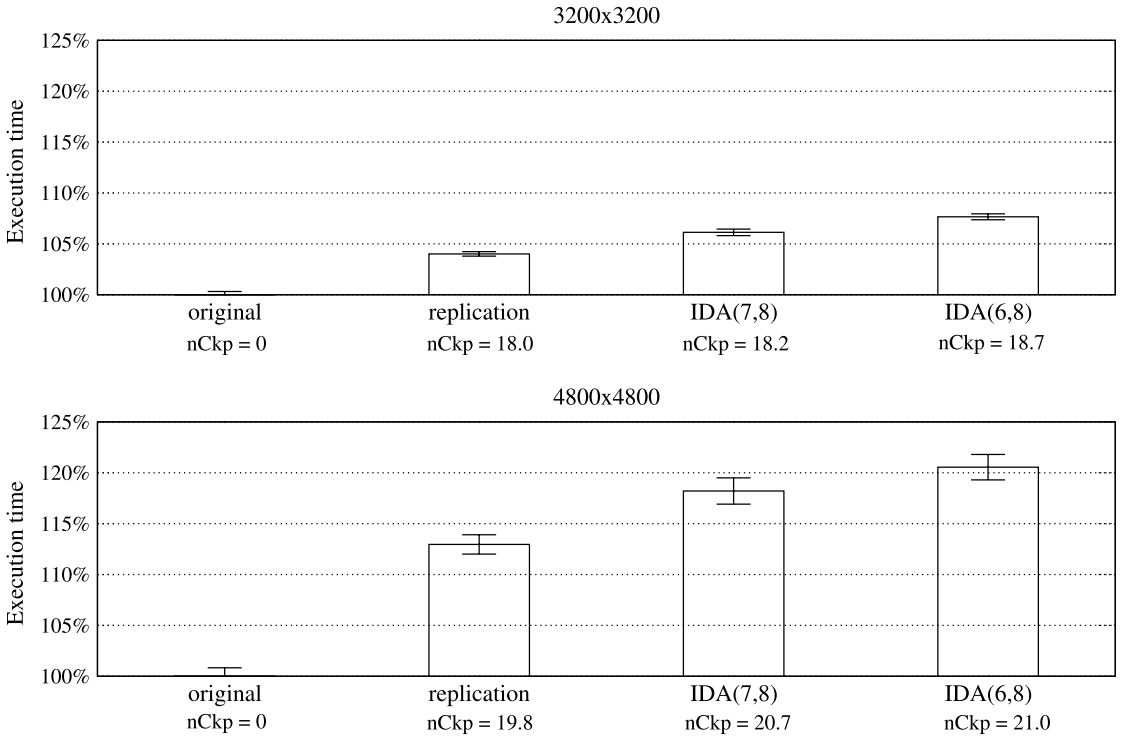

Fig. 6 Execution overhead of the checkpointing mechanism

large checkpoints. We configured the application to run in 8 nodes and used two different matrix sizes, 3200×3200 and 4800×4800, which generate global checkpoints of sizes 351.6 MB and 791.0 MB, respectively. The minimum interval between checkpoints in this experiment is 60 s. The cluster where the application was executed is composed of 9 1.46 GHz AthlonXP machines with 1 GB of RAM, running GNU/Linux 2.6 and connected by a switched 100 Mbps Fast Ethernet network.

In the replication scenario, our system stores one replica of the checkpoint in the machine running the process that generated the checkpoint and the other replica in a machine running another process of the application. We also used two IDA configurations. The first, IDA(7, 8), generates 8 fragments from which 7 are required to reconstruct the original file, and the second, IDA(6, 8), generates 8 fragments and requires 6 for file reconstruction.

Figure 6 shows the results of 16 executions for each strategy. For a matrix of size 4800×4800, which generates global checkpoints of 791 MB, and a minimum interval between checkpoints of 60 s, the overhead is 13% when using replication, 18% when using IDA(7, 8) and 20% for IDA(6, 8), when compared to the execution time of 1101.6 s without checkpointing. The small standard deviation was expected, since the cluster was used exclusively during our experiment. These results show that the usage of checkpointing has a low overhead even when running coupled applications that generate large checkpoints.

Using the IDA strategy causes a higher overhead, since it is necessary to code the checkpoints using the processor cycles from the machines running the parallel application. Using replication causes a smaller overhead, but uses more network and storage resources. Since parallel applications normally run on a single cluster connected by a local LAN and storage space is not a scarce resource, the default strategy used is replication. Also, the overhead can be greatly reduced by increasing the checkpointing interval. For example, with a checkpointing interval of 5 minutes, the overhead would be of approximately 2.6% using replication.

7 Conclusions

We presented an integrated middleware system that allows institutions to use idle resources from existing machines to carry out high-performance computing and store application data in a reliable way. Different from previous works in the field, our middleware allows the sharing of unused disk space, in addition to idle processor cycles and main memory.

Reliable execution of parallel applications in nondedicated resources is provided by a checkpointing mechanism with a low execution overhead, with the application input, output, and checkpointing data managed by specialized modules that store the data in the local cluster or distributed across several clusters. Regarding data storage, we showed

that we can achieve successful recovery rates above 99%, using a moderate level of redundancy, even in the presence of a high number of node departures and failures and using only the idle period of machines. Also, file storage and retrieval are efficient and take advantage of the fastest available connections. These results indicate that our integrated middleware successfully permits both the reliable execution of applications and the storage of application data.

As ongoing work, we will deploy our middleware for long periods and monitor its usage, validating results obtained in the simulations in a real deployment. When evaluating the machine availability, we should also consider disk usage to avoid penalizing machines that deal mostly with disk intensive applications. Finally, we will consider limiting the network usage when there are machine owners utilizing the local networks for their tasks. This is relevant in some cases because if users notice a significant loss in network performance, they will be unwilling to share their resources with the grid.

References

1. Antoniu G, Bougé L, Jan M (2005) Juxmem: An adaptive supportive platform for data sharing on the grid. Scalable Comput Pract Exp 6(3):45–55
2. Batten C, Barr K, Saraf A, Trepetin S (2002) pStore: A secure peer-to-peer backup system. Tech Rep MIT-LCS-TM-632, MIT LCS
3. Blackham B (2009) Cryopid page. http://cryopid.berlios.de/
4. Blake C, Rodrigues R (2003) High availability, scalable storage, dynamic peer networks: pick two. In: HotOS'03: Proc of the 9th workshop on hot topics in operating systems, USENIX
5. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426. doi:10.1145/362686.362692
6. Bolosky WJ, Douceur JR, Ely D, Theimer M (2000) Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. SIGMETRICS Perform Eval Rev 28(1):34–43. doi:10.1145/345063.339345
7. Bronevetsky G, Marques D, Pingali K, Stodghill P (2003) Automated application-level checkpointing of MPI programs. In: PPoPP '03: Proceedings of the 9th ACM, SIGPLAN symposium on principles and practice of parallel programming, pp 84–89
8. Cai M, Chervenak A, Frank M (2004) A peer-to-peer replica location service based on a distributed hash table. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on supercomputing. IEEE Computer Society, Washington, p 56. doi:10.1109/SC.2004.7
9. de Camargo RY, Kon F (2006) Distributed data storage for opportunistic grids. In: ACM/IFIP/USENIX middleware doctoral symp, Melbourne, Australia
10. de Camargo RY, Kon F (2007) Design and implementation of a middleware for data storage in opportunistic grids. In: Proceedings of the 7th IEEE international symposium on cluster computing and the grid (CCGRID 2007), Rio de Janeiro, Brazil. IEEE Computer Society, Washington, pp 23–30
11. de Camargo RY, Kon F, Goldman A (2005) Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments. In: SBAC-PAD'05: The 17th international symposium on computer architecture and high performance computing, Rio de Janeiro, Brazil
12. de Camargo RY, Castor Filho F, Kon F (2009) Efficient maintenance of distributed data in highly dynamic opportunistic grids. In: Proceedings of the 24th ACM symposium on applied computing (SAC 2009), Track on dependable and adaptive distributed systems (DADS), Honolulu, HI, USA. ACM, New York
13. Chervenak AL, Palavalli N, Bharathi S, Kesselman C, Schwartzkopf R (2004) Performance and scalability of a replica location service. In: HPDC '04: Proceedings of the 13th IEEE international symposium on high performance distributed computing (HPDC'04). IEEE Computer Society, Washington, pp 182–191. doi:10.1109/HPDC.2004.27
14. Chiba S (1995) A metaobject protocol for C++. In: OOPSLA '95: Proceedings of the 10th ACM conference on object-oriented programming systems, languages, and applications, pp 285–299
15. Chien A, Calder B, Elbert S, Bhatia K (2003) Entropia: architecture and performance of an enterprise desktop grid system. J Parallel Distrib Comput 63(5):597–610. doi:10.1016/S0743-7315(03)00006-6
16. Cirne W, Brasileiro F, Andrade N, Costa L, Andrade A, Novaes R, Mowbray M (2006) Labs of the world, unite!!! J Grid Comput 4(3):225–246
17. Dabek F, Kaashoek MF, Karger D, Stoica I Morris R (2001) Wide-area cooperative storage with cfs. In: SOSP '01: Proceedings of the eighteenth ACM symposium on operating systems principles. ACM, New York, pp 202–215. doi:10.1145/502034.502054
18. Domingues P, Marques P, Silva L (2005) Resource usage of windows computer laboratories. In: Proc of the int conf on parallel processing (ICCP'05): workshops, pp 469–476
19. Elnozahy M, Alvisi L, Wang YM, Johnson DB (2002) A survey of rollback-recovery protocols in message-passing systems. ACM Comput Surv 34(3):375–408
20. Goldchleger A, Kon F, Goldman A, Finger M, Bezerra GC (2004) InteGrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. Concurr Comput Pract Exp 16:449–459
21. Goldchleger A, Goldman A, Hayashida U, Kon F (2005) The implementation of the bsp parallel computing model on the integrade grid middleware. In: MGC '05: Proceedings of the 3rd international workshop on middleware for grid computing. ACM, New York, pp 1–6. doi:10.1145/1101499.1101504
22. Hayashida UK, Okuda K, Panetta J, Song SW (2005) Generating parallel algorithms for cluster and grid computing. In: ICCSA '05: The 2005 international conference on computational science and its applications. Springer, Berlin, pp 509–516
23. Karablieh F, Bazzi RA, Hicks M (2001) Compiler-assisted heterogeneous checkpointing. In: SRDS '01: Proceedings of the 20th IEEE symposium on reliable distributed systems, New Orleans, USA, pp 56–65
24. Kircher M, Jain P (2004) Pattern-oriented software architecture, Volume 3: patterns for resource management. Wiley, New York
25. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
26. Landers M, Zhang H, Tan KL (2004) Peerstore: Better performance by relaxing in peer-to-peer backup. In: P2P '04: Proc of the 4th int conf on peer-to-peer computing. IEEE Computer Society, Washington, pp 72–79. doi:10.1109/P2P.2004.38
27. Litzkow M, Livny M, Mutka M (1988) Condor—a hunter of idle workstations. In: ICDCS '88: Proceedings of the 8th int conference of distributed computing systems, pp 104–111
28. Luckow A, Schnor B (2008) Adaptive checkpoint replication for supporting the fault tolerance of applications in the grid. In: Proceedings of the 2008 seventh IEEE international symposium on network computing and applications. IEEE Computer Society, Washington, pp 299–306. doi:10.1109/NCA.2008.38
29. Malluhi QM, Johnston WE (1998) Coding for high availability of a distributed-parallel storage system. IEEE Trans Parallel Distrib Syst 9(12):1237–1252. doi:10.1109/71.737699

30. Mutka MW, Livny M (1991) The available capacity of a privately owned workstation environment. Perform Eval 12(4):269–284. doi:10.1016/0166-5316(91)90005-N
31. Plank JS, Kingsley MBG, Li K (1995) Libckpt: Transparent checkpointing under unix. In: Proceedings of the USENIX winter 1995 technical conference, pp 213–323
32. Plank JS, Li K, Puening MA (1998) Diskless checkpointing. IEEE Trans Parallel Distrib Syst 9(10):972–986. doi:10.1109/71.730527
33. Pruyne J, Livny M (1996) Managing checkpoints for parallel programs. In: IPPS '96: Proceedings of the workshop on job scheduling strategies for parallel processing. Springer, London, pp 140–154
34. Rabin MO (1989) Efficient dispersal of information for security, load balancing, and fault tolerance. J ACM 36(2):335–348. doi:10.1145/62044.62050
35. Ripeanu M, Foster I (2002) A decentralized, adaptive replica location mechanism. In: HPDC '02: Proceedings of the 11th IEEE international symposium on high performance distributed computing. IEEE Computer Society, Washington
36. Rowstron A, Druschel P (2001) Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In: SOSP '01: Proceedings of the eighteenth ACM symposium on operating systems principles. ACM, New York, pp 188–201. doi:10.1145/502034.502053
37. Rowstron AIT, Druschel P (2001) Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware 2001: IFIP/ACM international conference on distributed systems platforms, Heidelberg, Germany, pp 329–350
38. Sobe P (2003) Stable checkpointing in distributed systems without shared disks. In: IPDPS '03: Proceedings of the 17th international symposium on parallel and distributed processing. IEEE Computer Society, Washington, p 214.2
39. Stoica I, Morris R, Karger D, Kaashock M, Balakrishman H (2003) Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans Netw 11(1):17–32
40. Strumpen V, Ramkumar B (1996) Portable checkpointing and recovery in heterogeneous environments. Tech Rep UI-ECE TR-96.6.1, University of Iowa
41. Valiant L (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111
42. Vazhkudai SS, Ma X, Freeh VW, Strickland JW, Tammineedi N, Scott SL (2005) Freeloader: Scavenging desktop storage resources for scientific data. In: SC '05: Proceedings of the 2005 ACM/IEEE conference on supercomputing. IEEE Computer Society, Washington, p 56. doi:10.1007/s13173-010-0016-0
43. Weatherspoon H, Kubiatowicz J (2002) Erasure coding vs. replication: a quantitative comparison. In: IPTPS '01: Revised papers from the first international workshop on peer-to-peer systems. Springer, London, pp 328–338