IEEE distributed systems Expert-authored articles and resources ONLINE

September 2006 (vol. 7, no. 9), art. no. 0609-o9001
1541-4922 © 2006 IEEE
Published by the IEEE Computer Society

Data Storage

Strategies for Checkpoint Storage on Opportunistic Grids

Raphael Y. de Camargo and Fabio Kon • University of São Paulo
Renato Cerqueira • Pontifical Catholic University of Rio de Janeiro

This article evaluates several strategies for storing checkpoint data in an opportunistic grid environment, including replication, parity information, and erasure coding. This evaluation compares the computational overhead, storage overhead, and degree of fault tolerance of these strategies.

Executing computationally intensive parallel applications on dynamic heterogeneous environments such as computational grids can be a daunting task,[1-3] especially when using nondedicated resources. Such is the case for opportunistic computing,[4] where we use only the shared machines' idle periods. In such a scenario, machines often fail and frequently change their state from idle to occupied, compromising their execution of applications. Unlike dedicated resources, whose mean time between failures is typically weeks or even months, nondedicated resources can become unavailable several times during a single day. In fact, some machines are unavailable more than they're available. Fault tolerance mechanisms, such as checkpoint-based rollback recovery,[5] can help guarantee that applications execute properly amid frequent failures.

The fault tolerance mechanism must save generated checkpoints on a stable storage medium. The usual solution is to install checkpoint servers and connect them to the nodes through a high-speed network. But a dedicated server can easily become a bottleneck as grid size increases. Moreover, when using an opportunistic computing environment, relying on such dedicated hardware increases hardware costs and contradicts the objective of such a system, which is to use the idle time of shared machines. The simplest solution is to use the grid's shared nodes as the storage medium for checkpoints, thus storing and retrieving data in the nodes' shared disk space. But, to preserve the machine's quality of service, it's best to store and retrieve data from machines only when they are idle. Consequently, it's likely that data stored on a machine will be unavailable when requested to restart a failed application.

One way to solve this problem is to store multiple replicas of checkpoint data, so that you can recover the stored data even when part of the data repositories are unavailable. Another approach is to break data into several fragments, adding some redundancy, to enable data recovery from a subset of the fragments. Two common techniques for splitting data into redundant fragments are the use of erasure coding, such as information dispersal algorithms,[6] and the addition of parity information.

In this article, which builds on previously published work,[7] we evaluate several strategies for the distributed storage of checkpoint data in opportunistic environments. (The "Related Work" sidebar discusses other recent work in this area.) We focus on the storage of checkpoint data inside a single cluster. We present a prototype implementation of a distributed checkpoint repository over InteGrade,[8] a multiuniversity grid middleware project to leverage the computing power of idle shared workstations. Using this prototype, we performed several experiments to determine the trade-offs in these strategies between computational overhead, storage overhead, and degree of fault tolerance.

Data storage strategies

A data-coding strategy must consider scalability, computational cost, and fault tolerance. We analyze three different strategies for data coding: data replication, data parity, and erasure coding in an information dispersal algorithm. We briefly compare their computational cost, storage overhead, and effects on data availability.

To evaluate data availability, we must determine which machines will store checkpoint data. It's possible to distribute the data over the nodes executing the application, other grid nodes, or both. Here, we evaluate checkpoint data storage in the machines where the parallel application executes, using an extra node when the coding strategy requires one. We use this approach because a parallel application normally tries to use most of the available nodes in a cluster to execute its processes. This approach also lets us couple application failures with data repositories failures, making it easier to control checkpoint data availability at any time.

In this work, we don't address data integrity or privacy. It's possible to check data integrity using secure hash functions such as MD5 and SHA-1. Encrypting the checkpoints ensures data privacy. Data encryption is a computationally intensive process and should be used only when necessary.

Data replication

Using data replication, we store full replicas of the generated checkpoints. If one of the replicas becomes inaccessible, we can use another. The advantage is that no extra coding is necessary, but the disadvantage is that we must transfer and store large amounts of data. For example, guaranteeing safety against a single failure requires saving two copies of the checkpoint.

In our scenario, transferring two times the checkpoint data would generate too much network traffic, so we store a copy of the checkpoint locally and another remotely. Even though a failure in a machine running the application will make one of the checkpoints inaccessible, it will be 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.

Parity

An alternative to data replication is to slice the checkpoint data into several fragments and store these fragments with an additional fragment containing parity information. This avoids data replication's large storage requirements because it requires the storage of only one checkpoint copy.

We use a scheme in which each node calculates its checkpoint's parity locally, first dividing the generated checkpoint into m slices and then calculating the parity over these slices. We divide a checkpoint vector, $\boldsymbol{U}$, of size n into m slices of size n/m, given by

$\mathbf{U} = \mathbf{U}_0, \mathbf{U}_1, \ldots, \mathbf{U}_m$, and

$\mathbf{U}_k = u_0^k, u_1^k, \ldots, u_{n/m}^k, \; 0 \le k < m$

where k is the slice number, and u_i^k represents the elements of slice $\mathbf{U}_k$. We calculate elements p_i, $0 \le i < n/m$ of parity information vector $\mathbf{P}$ as

$p_i = u_i^0 \oplus u_i^1 \oplus \ldots \oplus u_i^m, \; 0 \le i < n/m$

where $\oplus$ represents the exclusive-or (XOR) operation. From each process, the system then distributes slices $\mathbf{U}_i$ and parity vector **P** for storage on other nodes. Similarly, we can recover a missing fragment by performing the XOR operation over the recovered fragments.

Evaluating parity is fast because it requires only simple XOR operations and storage overhead is very small. The drawback is that the parity strategy doesn't tolerate two or more simultaneous failures.

Information dispersal algorithm

Michael Rabin's classic information dispersal algorithm (IDA) generates a space-optimal coding of data.[6] It allows coding a vector **U** of size n into $m + k$ encoded vectors of size n/m, such that regenerating **U** is possible using only m encoded vectors. This encoding lets you achieve different fault tolerance levels by merely tuning the values of m and k. In practice, it's possible to tolerate k failures with an overhead of only $k/(mn)$ elements.

This algorithm requires the computation of mathematical operations over a Galois field GF(q), a finite field of q elements, where q is either prime or a power p^x of prime number p. When using $q = p^x$, you carry arithmetic operations over the field by representing the numbers as polynomials of degree x and coefficients in [0, p – 1]. You calculate sums with XOR operations, whereas you carry out multiplications by multiplying the polynomials modulo an irreducible polynomial of degree x. In our case, we use $p = 2$ and $x = 8$, representing a byte. To speed up calculations, we perform simple table lookup for the multiplications.

The algorithm also requires the generation of $m + k$ linearly independent vectors $\mathbf{a}_i$ of size m. We can easily generate these vectors by choosing n distinct values a_i, $0 \le i < n$, and setting $\boldsymbol{\alpha}_i = (i, \mathbf{a}_i, \ldots \mathbf{a}_i^{n-1})$, $0 \le i < n$. We then organize these vectors as a matrix, **G**, defined as

$$\mathbf{G} = [\boldsymbol{\alpha}^T_0, \boldsymbol{\alpha}^T_1, \ldots \boldsymbol{\alpha}^T_{m+k}]$$

where T indicates the transpose of vector $\boldsymbol{\alpha}$. We now break file F into n/m information words, $\mathbf{U}_i$, of size m and generate n/m code words V of size $m + k$, where

$$\mathbf{V}_i = \mathbf{U}_i \times \boldsymbol{G}$$

The $m + k$ encoded vectors, $\mathbf{E}_i$, $0 \le i < m + k$, are given by

$$\mathbf{E}_i = \mathbf{V}_0[i], \mathbf{V}_1[\mathrm{i}], \ldots \mathbf{V}_{n/m}[\mathrm{i}]$$

To recover the original information words, $\mathbf{U}_i$, we need to recover k of the encoded $m + k$ slices. We then construct code words $\mathbf{V}_j'$, which are equivalent to the original code words $\mathbf{V}_i$ but contain only the components of the k recovered slices. Similarly, we construct matrix $\boldsymbol{G'}$, containing only elements relative to the recovered slices. We now recover $\mathbf{U}_i$, multiplying encoded words $\mathbf{V}_j'$ by the inverse of $\mathbf{G'}$:

$$\mathbf{U}_\mathrm{i} = \mathbf{V}_j' \times (\boldsymbol{G'})^{-1}$$

The main drawback of this approach is that coding requires $O[(m + k)nm]$ steps and decoding requires $O(nm^2)$ steps, in addition to the inversion of an $m \times m$ matrix. Qutaibah Malluhi and William Johnston proposed an algorithm that improves coding computation complexity to $O(nmk)$ and also improves decoding.[9] They showed that you can diagonalize the first m columns of $\boldsymbol{G}$ and still have a valid algorithm. Consequently, the first m fields of code words $\mathbf{V}_i$ involve simple data copying. Coding is necessary only for the last k fields. This approach reduces encoding complexity considerably.

IDA's greatest advantage is that it provides the desired degree of fault tolerance with optimal space overhead. For an application composed of 10 nodes, if we set *m* as 10, the algorithm can tolerate a failure of one node with a 10 percent space overhead, two failures with a 20 percent overhead, and so on. The disadvantage of this approach is the computational complexity of implementing the algorithm and the higher computational overhead.

Case study: InteGrade

InteGrade is a grid middleware solution for harnessing idle computing power from shared workstations.[7] It consists of a collection of hierarchically organized InteGrade clusters. Here, we focus on checkpoint storage inside a single cluster.

Architecture

Figure 1 shows an InteGrade cluster's main modules. The global resource manager (GRM) controls resource management at the cluster level; the local resource manager (LRM) controls it at the node level. To form a grid composed of a cluster federation, GRMs from different clusters communicate with one another to allow global sharing of local resources. An LRM communicates only with modules from its own cluster. This separates resources belonging to different clusters; consequently, administrators can apply custom policies for each cluster.

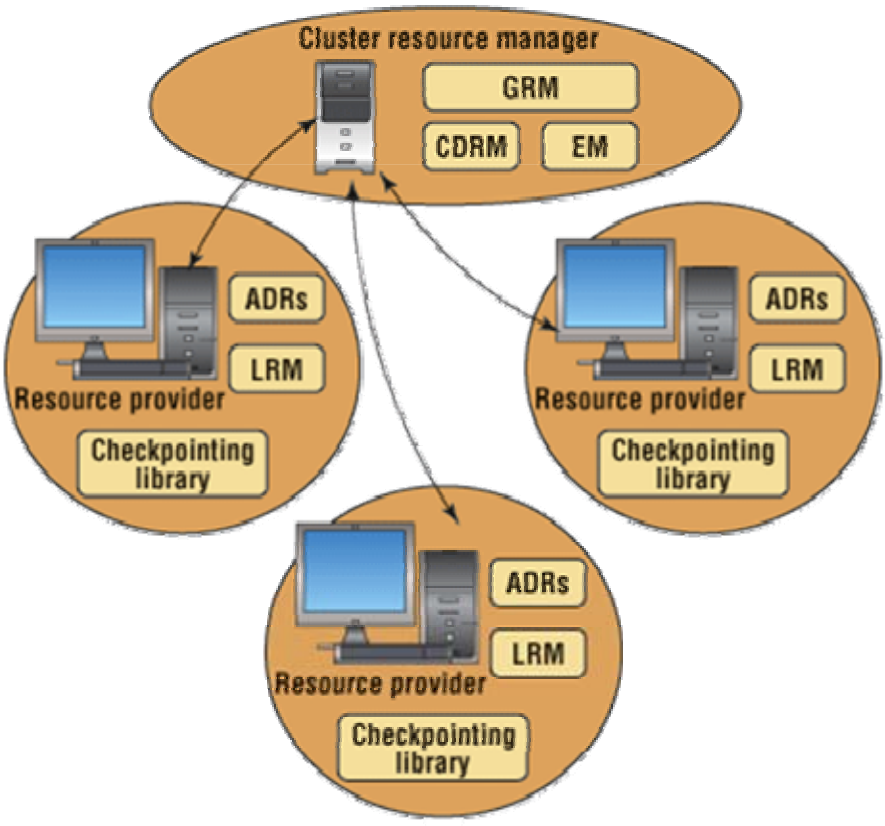

Figure 1. InteGrade's intracluster architecture.

InteGrade provides portable application-level checkpointing and rollback recovery for sequential applications and for BSP (bulk synchronous parallel) and parameter-sweeping parallel applications.[10] The main modules for storing checkpoint data are the checkpointing library, the execution manager (EM), the cluster data repository manager (CDRM), and the autonomous data repositories (ADRs).

The checkpointing library provides the functionality to periodically generate portable checkpoints containing the application state. The EM maintains a list of applications executing on the cluster and coordinates the reinitialization process when an application fails. The CDRM manages the available ADRs from its cluster and the location of checkpoint data fragments. The ADRs store checkpoint data; they reside on machines that share their resources with the grid.

When the checkpointing library needs to store checkpoint data, it queries the local CDRM for available local data repositories and then transfers the data to the returned repositories. Checkpoint data recovery involves the library querying the CDRM for the list of repositories containing the checkpoints and retrieving the checkpoints from these repositories.

The benefits of using a centralized GRM, EM, and CDRM for each cluster include implementation ease and simpler algorithms requiring less message exchanges. Moreover, these modules are part of a cluster federation, thus allowing replication and logging of their contents in other clusters for increased fault tolerance.

Experiments

We performed the experiments using 11 AthlonXP 1700+ processors with 1 Gbyte of RAM, connected by a switched 100-Mbps Fast Ethernet network. Students use the machines during the day, so we performed the experiments at night. Our objective was to measure the overhead of checkpoint storage during normal operation without machines becoming unavailable. We configured all machines as part of a single InteGrade cluster.

We used a matrix multiplication application with matrices of different sizes and composed of 12-byte double-precision elements. We evaluated the time necessary to encode and decode data and the overhead of using different storage strategies.

Data encoding and decoding

We first measured the time required to encode and decode a checkpoint using IDA and local parity. We varied the data size and the number of slices for the comparison. Figure 2 shows the results. In the graphs, IDA (*m*, *k*) represents IDA using *m* and *k* as described earlier.

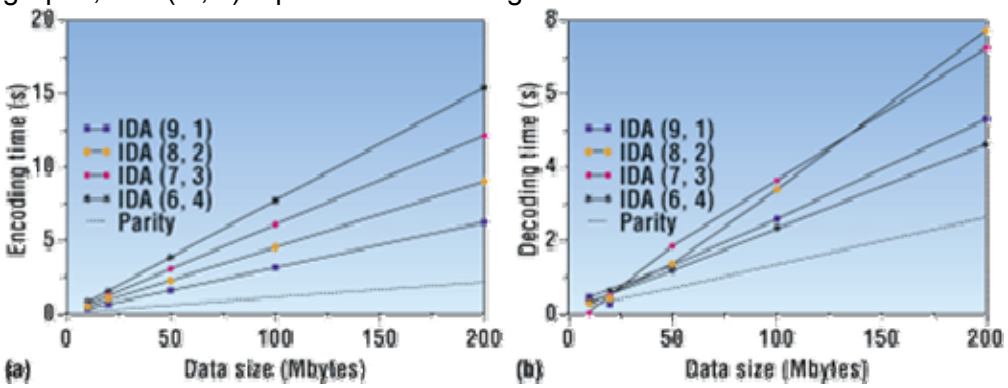

Figure 2. Time required to (a) code and (b) decode a file.

As expected, the local parity calculation was faster than IDA in all scenarios. The most interesting result, however, was that coding with IDA wasn't too expensive. Encoding 100 Mbytes of data required only a few seconds; the encoding time increased linearly with the number of extra slices *k* and the data size. The same was true for decoding. Therefore, recovering the data shouldn't take more than a few seconds. With further optimizations in vector multiplications, we can achieve even better results. Consequently, the results of this experiment were very satisfactory.

Execution overhead

We also evaluated the overhead incurred by checkpointing, coding, and distributing storage over a parallel-application execution time. The objective was to compare the overhead for several of the storage strategies we described earlier. We evaluated the following scenarios:

- *No storage*. The system generates checkpoints but doesn't store them.
- *Centralized repository*. The system stores checkpoints in a centralized repository.
- *Replication*. The system stores one copy of the checkpoint locally and another in a remote repository.
- *Parity over local checkpoints*. The system breaks the checkpoint into 10 slices, with one containing parity information, and stores them in distributed repositories.
- *IDA* ($m = 9$, $k = 1$). The system codes the checkpoint into 10 slices, from which nine are sufficient for recovery, and stores them in distributed repositories.
- *IDA* ($m = 8$, $k = 2$). The system codes the checkpoint into 10 slices, from which eight are sufficient for recovery, and stores them in distributed repositories.

When using replication, the system distributes remotely stored checkpoints throughout the nine nodes executing the application. For the last three scenarios, which generate 10 slices, we used an additional node to store the remaining slice. We stored the checkpoints in the machines executing the application processes so that we could evaluate how checkpoint storage affects application execution time. If other machines in the cluster are idle, it would be better to store data on these machines, thus transferring the storage overhead to them.

Figure 3 compares the overhead of storing checkpoints to the overhead when the application generates but doesn't store checkpoints. The x-axis contains the six storage scenarios. The y-axis shows the normalized execution time. We used nine nodes to perform the matrix multiplication and three matrix sizes: 1,350 × 1,350; 2,700 × 2,700; and 5,400 × 5,400. To perform the benchmark, we divided the total execution time into execution segments bounded by the checkpoint generation times. Table 1 gives these values for each matrix, along with the number of generated checkpoints and the size of local and global checkpoints.

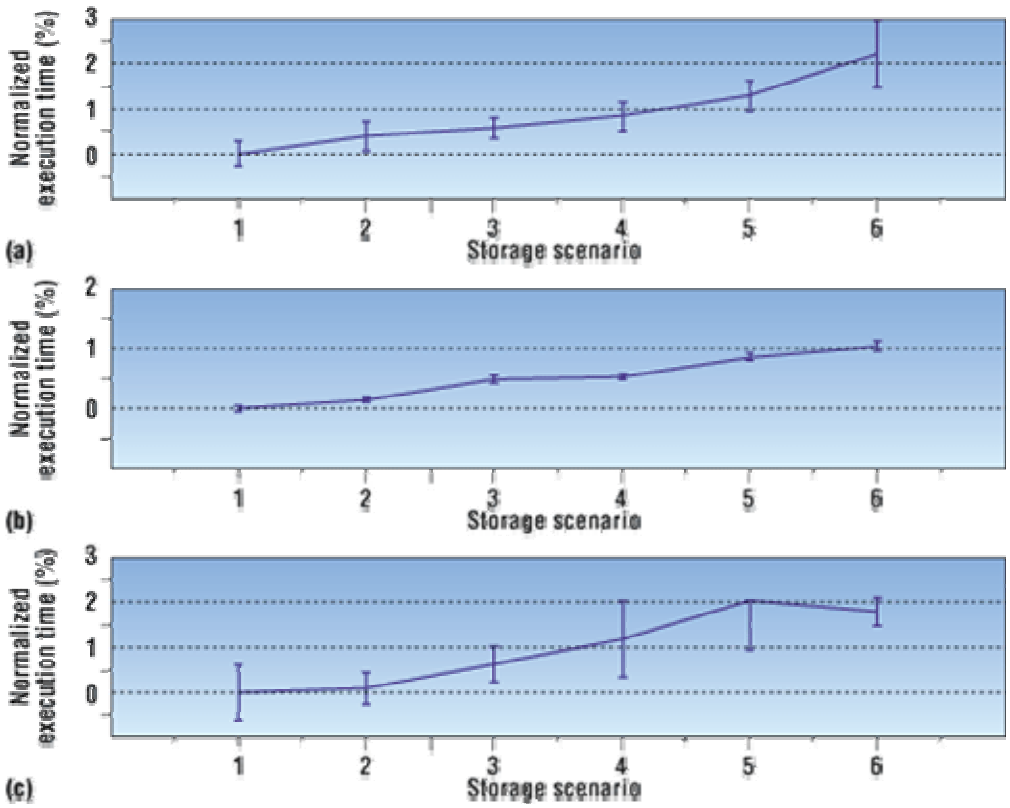

Figure 3. Checkpoint storage overhead for the matrix multiplication application: (a) 1,350 × 1,350 matrix, (b) 2,700 × 2,700 matrix, and (c) 5,400 × 5,400 matrix.

Table 1. Execution parameters for the execution overhead experiment.

Matrix size	Total execution time (s)	Segment mean execution time (s)	No. of generated checkpoints	Size of local checkpoints (Mbytes)	Size of global checkpoints (Mbytes)
1,350 × 1,350	908.5	54.8	17	7.3	65.6
2,700 × 2,700	2,117.0	303.1	7	29.2	262.4
5,400 × 5,400	4,281.6	616.1	7	116.6	1,047.7

The results show that IDA incurs the highest overhead—which we expected, because IDA requires data encoding. But the extra overhead was always below 3 percent, which is very small, especially considering the large checkpoint sizes. Although for the 5,400 × 5,400 matrices the checkpoint interval was 10 minutes, we could reduce this value to 5 minutes or less and still get a reasonable overhead.

In comparing these storage strategies, we saw that using parity, replication, or centralized storage could reduce overhead, but with a lower degree of fault tolerance. Encoding with IDA was slower but more flexible, because by manipulating its parameters we could trade off between speed, resource use, and level of fault tolerance. Moreover, IDA requires less storage space and network use, thus allowing better resource utilization. Finally, in our experimental scenario, the execution overhead of producing, coding, and storing global checkpoints greater than 1 Gbyte was small, always below 3 percent, for typical checkpoint intervals of a few minutes.

Storing all checkpoint data inside a single cluster is efficient because machines are normally connected by fast switched networks. But this approach has the drawback of being sensitive to correlated failures in the cluster nodes. A scenario in which all machines in a cluster become unreachable is quite common and can occur, for example, from a problem in the local network.

Because we're dealing with a federation of clusters, we can make the system tolerant to correlated failures by distributing the checkpoint fragments randomly throughout the grid. IDA seems to be a good choice for encoding data because it lets us code a file into an arbitrary number of fragments and requires only of a subset of them to recover the original file. For example, we could encode a file into 32 blocks, from which 16 are required for recovery, to achieve very high availability levels.[11,12]

The main problem with using remote clusters for storage is that sending and fetching data from distant repositories is typically far slower than operating in the local cluster. A good solution is to store most of the generated checkpoints in the local cluster and the remaining ones in remote repositories. For example, for each five generated checkpoints, the system would store four in the local cluster and one in remote clusters. This solution would prevent the large overhead of sending data through long distances, at the cost of possibly greater computation loss if there are correlated failures in the cluster nodes.

However, coordinating data distribution in the entire grid is complex, requiring the clusters to communicate with one another and share information about their available data repositories and locally stored files. Several challenges remain for efficient distributed storage, including the development of scalable algorithms for data location, data consistency, data privacy, and fault tolerance.

We are working on a distributed storage system for computational grids that allows reliable storage of arbitrary data. In this scheme, CDRMs from InteGrade clusters communicate with one another using a structured peer-to-peer overlay network. The system encodes data using IDA and then distributes the fragments throughout the grid. Each fragment receives a unique ID, which the system uses to route the fragment to the target CDRM. This system stores all data related to grid applications, including input, output, and checkpoint data.

Acknowledgments

A grant from CNPq, Brazil (process no. 55.2028/02-9) supported this work.

References

1. F. Berman, G. Fox, and T. Hey, *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley & Sons, 2003.
2. R.Y. de Camargo et al., "The Grid Architectural Pattern: Leveraging Distributed Processing Capabilities," *Pattern Languages of Program Design 5*, D. Manolescu, J. Noble, and M. Völter, eds., Addison-Wesley, 2006, pp. 337–356.
3. I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 2003.
4. M. Litzkow, M. Livny, and M. Mutka, "Condor—A Hunter of Idle Workstations," *Proc. 8th Int'l Conf. Distributed Computing Systems* (ICDCS 88), IEEE CS Press, 1988, pp. 104–111.
5. M. Elnozahy et al., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, 2002, pp. 375–408.
6. M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," J. ACM, vol. 36, no. 2, 1989, pp. 335–348.
7. R.Y. de Camargo, R. Cerqueira, and F. Kon, "Strategies for Storage of Checkpointing Data Using Non-Dedicated Repositories on Grid Systems," *Proc. 3rd Int'l Workshop Middleware for Grid Computing* (MGC 05), ACM Press, 2005, pp. 1–6.

8. A. Goldchleger et al., “InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines,” *Concurrency and Computation: Practice and Experience*, vol. 16, no. 5, 2004, pp. 449–459.
9. Q.M. Malluhi and W.E. Johnston, “Coding for High Availability of a Distributed-Parallel Storage System,” *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 12, 1998, pp. 1237–1252.
10. R.Y. de Camargo, F. Kon, and A. Goldman, “Portable Checkpointing and Communication for BSP Applications on Dynamic Heterogeneous Grid Environments,” *Proc. 17th Int’l Symp. Computer Architecture and High Performance Computing*, IEEE CS Press, 2005, pp. 226–234.
11. R. Rodrigues and B. Liskov, “High Availability in DHTs: Erasure Coding vs. Replication,” *Proc. 4th Int’l Workshop Peer-to-Peer Systems* (IPTPS 05), Springer, 2005, pp. 226–239.
12. H. Weatherspoon and J. Kubiatowicz, “Erasure Coding vs. Replication: A Quantitative Comparison,” *Proc. 1st Int’l Workshop Peer-to-Peer Systems* (IPTPS 02), Springer, 2002, pp. 328–338.

Raphael Y. de Camargo is a doctoral candidate in the Department of Computer Science at the University of São Paulo. His research interests include grid computing, fault tolerance, distributed storage, and distributed-object systems. He received his master’s degree in physics from the University of São Paulo. Contact him at Rua Doutor Monteiro Tapajós, 70, São Paulo SP, Brazil, 04152040; rcamargo@ime.usp.br.

Renato Cerqueira is a research scientist and a project manager at the Computer Graphics Technology Group of the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), where he is also an assistant professor of computer science. His research interests include component-based development, object-oriented languages, middleware platforms, distributed programming, ubiquitous computing, and grid computing. He received his PhD in computer science from PUC-Rio. He’s a member of the ACM and IEEE Computer Society. Contact him at Computer Science Dept., PUC-Rio, Rua Marques de São Vicente 225, 407RDC, Rio de Janeiro RJ, Brazil 22453900; rcerq@inf.pucrio.

Fabio Kon is an associate professor in the Department of Computer Science at the University of São Paulo. His research interests include distributed-object systems, reflective middleware, dynamic reconfiguration and adaptation, mobile agents, computer music, multimedia, grid computing, and agile methods. He received his PhD in computer science from the University of Illinois at Urbana-Champaign. He’s a member of the ACM and the Hillside Group. Contact him at Departamento de Ciência da Computação, Rua do Matão, 1010, São Paulo SP, Brazil, 05508090; kon@ime.usp.br.

Related Links

- DS Online's Grid Computing Community
- "Portable Checkpointing and Communication for BSP Applications on Dynamic Heterogeneous Grid Environments," Proc. SBAC-PAD 05
- "Skewed Checkpointing for Tolerating Multi-Node Failures," Proc. 23rd IEEE In'tl Symp. Reliable Distributed Systems (SRDS 04)

Cite this article:
Raphael Y. de Camargo, Renato Cerqueira, and Fabio Kon, "Strategies for Checkpoint Storage on Opportunistic Grids," *IEEE Distributed Systems Online*, vol. 7, no. 9, 2006, art. no. 0609-o9001.

Sidebar: Related Work

Researchers have also compared several data storage strategies in different contexts. Hakim Weatherspoon and John Kubiatowicz compare erasure coding to replication in the context of peer-to-peer systems,[1] as do Rodrigo Rodrigues and Barbara Liskov.[2] In both cases, they evaluate the availability properties of erasure coding and replication using analytical formulations and collected data analysis. Our work focuses on the application execution overhead from coding and storing checkpoint data.

Peter Sobe analyzes two different parity techniques for storing checkpoints in distributed systems.[3] He compares the two models in an analytical study, but, unlike our work, this project doesn't include any experiments. Also, he evaluates a different set of storage strategies than we evaluate here.

James S. Plank, Kai Li, and Michael A. Puening propose the use of diskless checkpointing,[4] which involves storing checkpoint data on system-volatile memory, removing the overhead of stable storage. Like us, they evaluate strategies for storing checkpoint data on the processing nodes and one or more backup nodes. But the focus of their work is on comparing diskless and disk-based checkpointing, and they performed their experiments using parity information for fault tolerance.

Qutaibah Malluhi and William Johnston use an optimized version of Michael Rabin's information dispersal algorithm[5] and of 2D parity coding schemes, comparing their efficiency analytically.[6] We compare a different set of coding techniques, perform experimental evaluations, and focus on nondedicated repositories.

Finally, Jim Pruyne and Miron Livny study the use of multiple-checkpoint servers to store checkpoints from parallel applications, but they only compare single and dual dedicated checkpoint servers.[7]

References

1. H. Weatherspoon and J. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," *Proc. 1st Int'l Workshop Peer-to-Peer Systems* (IPTPS 02), Springer, 2002, pp. 328–338.
2. R. Rodrigues and B. Liskov, "High Availability in DHTs: Erasure Coding vs. Replication," *Proc. 4th Int'l Workshop Peer-to-Peer Systems* (IPTPS 05), Springer, 2005, pp. 226–239.
3. P. Sobe, "Stable Checkpointing in Distributed Systems without Shared Disks," *Proc. 17th Int'l Symp. Parallel and Distributed Processing* (IPDPS 03), IEEE CS Press, 2003, pp. 214–221.
4. J.S. Plank, K. Li, and M.A. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 10, 1998, pp. 972–986.
5. M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," *J. ACM*, vol. 36, no. 2, 1989, pp. 335–348.
6. Q.M. Malluhi and W.E. Johnston. "Coding for High Availability of a Distributed-Parallel Storage System," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 12, 1998, pp. 1237–1252.

7. J. Pruyne and M. Livny, “Managing Checkpoints for Parallel Programs,” *Proc. Workshop Job Scheduling Strategies for Parallel Processing* (IPPS 96), Springer, 1996, pp. 140–154.