Ensuring Application Continuity with Fault Tolerance Techniques

Rafaela Brum, Luan Teylo, Luciana Arantes, and Pierre Sens

Abstract A cloud is an attractive environment for executing high-performance (HPC) applications. There is an extensive and consolidated history of long-running HPC applications that were deployed on clouds or successfully migrated from clusters to clouds not only because the latter provides flexibility and access to virtually infinite resources but also because clouds are offered to users as failure-free platforms. However, outages are not uncommon in clouds and, in this case, the cloud provider and/or HPC applications need to implement fault tolerance mechanisms in order to ensure reliability and the correct execution of the applications. In this chapter, we present an overview of the related literature about fault tolerance (FT) techniques most used by clouds and HPC applications that run on them, basically checkpoint-rollback and replication, as well as fault detection approaches and existing reliable storage in clouds.

1 Introduction

A cloud environment is a distributed system composed of hundreds to millions of components. With such a scale, the probability of failures is extremely high and, therefore, failures become the norm and not the exception [97]. In general, public providers like Google and Amazon offer guarantees of high availability for their services but they are not 100% failure safe. For instance, in 2017, during an operational check-in on AWS, a typo in one of the commands executed by the

Rafaela Brum
Fluminense Federal University, Niterói, Brazil e-mail: rafaelabrum@id.uff.br

Luan Teylo
Inria Bordeaux Sud Ouest, Bordeaux, France e-mail: luan.gouveia-lima@inria.fr

Luciana Arantes, Pierre Sens
Sorbonne Université, CNRS, Inria, LIP6, Paris, France e-mail: {luciana.arantes, pierre.sens}@lip6.fr

technicians took down a great number of servers, affecting the S3 service. Suddenly, numerous services on the internet, including services offered by big players, like Quora and Spotify, started to report crashes. Millions of users could not use the services in question during the four hours needed to solve the problem [57]. Another recent incident happened in December 2020, when the N. Virginia region (*us-east-1*) of AWS EC2 faced a significant outage that took down lots of sites on the internet, rendering unavailable services that were primordial for the functionality of autonomous vacuum cleaners and doorbells [25]. Other public cloud providers, such as Microsoft Azure and Google Cloud, also had to cope with failures in the last years [4].

Therefore, the probability that failures have an impact on cloud and user applications is extremely high. In particular, in HPC applications, usually composed of long-running tasks whose execution can stand for days or even months, failures can have strong negative consequences on the correct execution of the application and even work loss. Therefore, it is fundamental to know which failure tolerance level a given HPC application requires and which is the ideal fault-tolerance technique to achieve it.

Beyond failures, a client or application tasks can suffer from interruptions related to the execution model of a cloud service. In this case, the provider does not offer full guarantees in terms of reliability and might revoke the service. The most well-known example of such services is the preemptible or spot VMs offered by the majority of cloud providers [32]. Such VMs have economic advantages but can be revoked at any time, contrary to on-demand VMs. The former can have prices up to 90% below the latter. One can argue that such revocations are not failures since they are not generated by unexpected behavior or bad functionality. However, fault tolerance techniques are mandatory for ensuring reliability of such services, otherwise applications will not work correctly. Hence, in this chapter, we also consider service revocation as a type of failure and discuss how FT techniques can be used to extract the maximum economic advantages of this execution model also guaranteeing the correct execution of applications.

HPC applications are typically executed in the cloud using the Infrastructure as a Service (IaaS) model where computational resources, such as storage, computing, and network, are offered as virtual machines (VMs) and on a pay-as-you-go basis [52]. Hence, in order to execute an application, the user requests a set of VMs, sets up the execution environment, and launches the application. At the end of the execution, the total monetary cost is computed based on the execution time and the VMs' costs. Nevertheless, one or more VMs can be interrupted due to failures or revocation, stopping the user's application and increasing the execution time and, in some cases, the monetary cost. Fault tolerance mechanisms play, thus, two fundamental roles: they ensure that the applications finish correctly and avoid high monetary cost increases.

In this chapter, we consider the two main types of faults: crash and resource revocation. In the context of clouds, a crash happens when a resource stops unexpectedly. For instance, when an on-demand VM stops working before the client releases it. On the other hand, a revocation happens when a resource, such as spot or preemptible

VM, is intentionally stopped by the provider. In both cases, the most common techniques used to tolerate them are checkpoint-rollback and replication. Therefore, we present an overview of solutions of the related literature that provides fault tolerance for HPC applications in cloud environments based on these two techniques. We also discuss some works in fault detection and different existing cloud reliable storage.

The remainder of the chapter is organized as follows. The next section summarizes fault tolerance techniques in distributed systems, focusing mainly on fault detection, checkpoint, and replication. Section 3 discusses the implementation of these techniques in clouds and the different approaches available in the related literature. Section 4, concludes the chapter and discusses some future directions and challenges.

2 Fault Tolerance

This section presents some fault-tolerant concepts and mechanisms existing in distributed systems. Section 2.1 discusses the concept of failure detection, a step before using the fault tolerance techniques while Section 2.2 and Section 2.3 respectively present the checkpoint/ recovery and replication techniques. Section 2.4 present some MPI projects and approaches that provide fault tolerance while Section 2.5 discusses some issues on applying fault tolerance mechanisms on HPC applications.

2.1 Failure Detection

A classic approach for tolerating failures in distributed systems is the detection and then the recovery of them. The failure detection phase is essential in reducing system unavailability, playing, thus, a central role in the engineering of such systems.

Proposed by Chandra and Toueg [21], unreliable failure detectors (FDs) can be seen as oracles which provide information on task crashes. They usually output a list of tasks suspected of having crashed. The information is unreliable in the sense that correct tasks might be falsely suspected of having crashed, and faulty tasks might still be trusted after they crashed. If an FD detects its mistake later, it corrects it. For instance, an FD can stop suspecting at time t + 1, a task that it suspected at time t. Unreliable failure detectors are usually characterized by two properties: completeness and accuracy, as defined by Chandra and Toueg [21]. Completeness characterizes the failure detector's capability of suspecting faulty tasks, while accuracy characterizes the failure detector's capability of not suspecting correct tasks, *i.e.*, restricts the mistakes that the failure detector can make. Two kinds of completeness and four kinds of accuracy are defined by Chandra and Toueg [21], which once combined yield eight classes of failure detectors.

Numerous failure detector implementations and classes have been proposed in the literature based on Chandra and Toueg's seminal work. They usually differ in the system assumptions such as type of node (identifiable, anonymous [10], homonymous [7]), type of link [1, 2, 47] (lossy asynchronous, reliable, timely, eventually timely, *etc.*), behavior properties [1, 54]; type of network (static [9, 47], dynamic [6, 35]), *etc.*

Regarding implementation, unreliable FDs usually exploit either a timer-based or a message-pattern approach. In the first one, FD implementations make use of timers to detect failures in tasks. Two mechanisms can be used to implement the timer-based strategy: heartbeat and pinging. In the heartbeat [23], every task q periodically sends an "I am alive" message to task p that is responsible for monitoring q. If p does not receive such a message from q after the expiration of a timer, it adds q to its list of suspected tasks. If p later receives an "I am alive" message from q, p then removes q from its list of suspected tasks. In the pinging mechanism [26, 100], every task p periodically sends a query message "Are you alive?" to the other tasks. Upon reception of such a message, a task q replies with an "I am alive" message. The heartbeat strategy has advantages over pinging since the former sends half of the messages than the latter for providing the same detection quality. Furthermore, a heartbeat detector estimates only the transmission delay of "I am alive" messages, whereas the pinging detector must estimate the transmission delay of "Are you alive?" messages, the reaction delay, and the transmission delay of "I am alive" messages.

The message-pattern strategy does not use any timeout mechanism. In Mostefaoui *et al.* [54], the authors propose an implementation that exploits such a strategy. A task p sends a QUERY message to n nodes that it monitors and then waits for responses (RESPONSE message) from α tasks ($\alpha \leq n$, traditionally $\alpha = n - f$, where f is the maximum number of failures). task p starts then to suspect every task that does not respond among the α first ones.

2.2 Checkpointing

Checkpointing and rollback recovery are well-known techniques to provide fault tolerance for parallel applications [3, 22, 30]. Each application task periodically saves its state on reliable storage in a checkpoint and, when a failure is detected, the execution is rolled back and resumed from earlier checkpoints. In a distributed context, backward error recovery of a task can result in a domino effect: to recover from a failure, the execution must be rolled back to a consistent state, but rolling back one task could result in an avalanche of rollbacks of other tasks before a consistent state is found. Figure 1 illustrates such an effect.

Numerous approaches to checkpointing and rollback recovery have been proposed in the literature for parallel systems. Checkpointing techniques can be divided into two categories: consistent and independent checkpointing.

With consistent checkpointing, tasks coordinate their checkpointing actions such that the collection of checkpoints represents a consistent state of the whole system where the saved local state of each task does not depend on the receipt of a message that is yet to be sent [30]. When a failure occurs, the system restarts from these

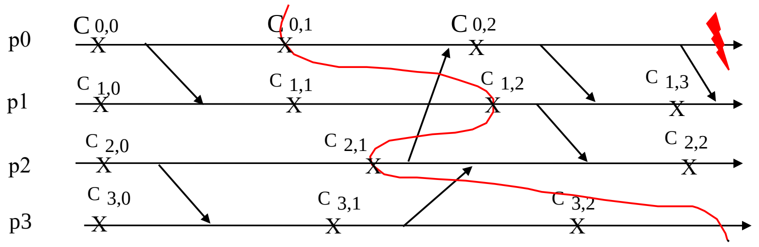

Fig. 1 Cascade of recoveries. Xs represent checkpoints. To recover from $p0$ failure, tasks $p0$, $p1$, and $p3$ need to recover from $C_{0,1}$, $C_{1,2}$, and $C_{2,3}$ to maintain a consistent global state represented by the red line.

checkpoints. Chandy and Lamport [22] proposed the first algorithm to save a consistent global state, assuming FIFO communication channels. When a task starts a new checkpoint, it sends a special message called marker over all its output channels. When a task receives a marker for the first time, it checkpoints. After beginning a checkpoint, all messages received from a neighbor n are added to the checkpoint image, until the marker reception from n. Figure 2 illustrates the Chandy-Lamport algorithm.

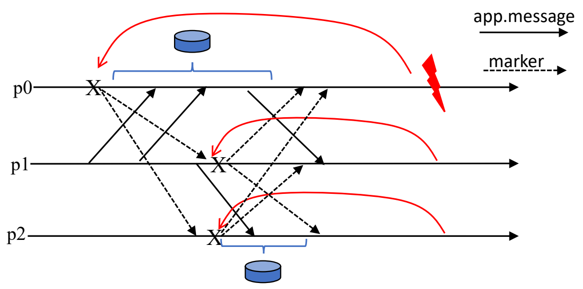

Fig. 2 Chandy-Lamport algorithm. Xs represent a consistent global state and messages "in transit" are logged. In case of failure, all tasks recover only from their last checkpoint.

The main drawback of this approach is that the messages used for synchronizing a checkpoint are an important source of overhead. Moreover, after a failure, surviving tasks may have to rollback to their latest checkpoint in order to remain consistent with recovering tasks. Alternatively, Koo and Toueg [45] reduce the number of tasks to rollback, by analyzing the interactions between tasks.

In the second approach, each task independently saves its state with no synchronization with the others. This technique is simple, but since the set of checkpoints may not define a consistent global state, the failure of one task leads to the rollback of other tasks. A reliable message logging [13, 27] avoids this domino effect. Logging methods fall into two classes: pessimistic and optimistic. Pessimistic message logging synchronously saves messages [16, 75], *i.e.*, the receiver is blocked until the message is logged on stable storage. In this way, all sent messages are logged, and a task in its recovered execution will directly access the log to receive again messages in the same order. A recovered task has then no interaction with the others until it reaches the last state before the failure. The optimistic message logging reduces failure-free overhead by logging recovery information asynchronously [16, 89]. Several messages can be grouped together and written to the stable storage in a single operation to reduce the logging overhead. However, tasks that survive a failure may be rolled back.

Another central concern when implementing a checkpointing technique is the involved time overheads. Basically, they can be divided into *recovery* and *dump*. The latter concerns the time spent recording on stable storage the application state in checkpoint files [95] while the former is related to the time spent reading these files and restarting the application. Depending on the recovery approach, the recovery time can also include extra overheads, such as the time to detect the failure. Consequently, both time overheads have a direct impact on the efficiency of the checkpointing technique.

In order to illustrate the difficulty in choosing a good checkpointing strategy, let's consider the example in Fig. 3, where two different execution scenarios are presented. In both cases, the application records a series of checkpoints, and each of them takes 5 time units to be recorded. In the first scenario, no failure happens, and the total execution time of the application is 75 time units. In the second scenario, just after the second checkpointing, the platform faces a failure (represented by the red x) and the application is interrupted. Once the failure is detected, the recovery task starts, and the application rolls back to its last record state, finishing with a total execution time of 95 units.

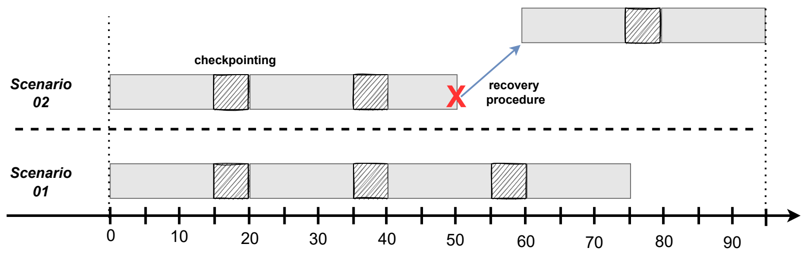

Fig. 3 Execution scenarios illustrate the checkpoint and recovery approach

In both scenarios, the total dump time was 15 units of time (5 units per checkpointing). In Fig. 3, the recovery took 10 time units.

Sometimes, the advantages of using a checkpoint strategy are not straightforward. For instance, considering the total execution time without any monetary cost, if a failure takes place in period 50, an application without checkpointing will restart from the beginning, spending 110 units of time. Thus, in this case, the checkpoint is worthwhile. However, if the dump time was 15 time units instead of 5, restarting the application from the beginning would spend less time. Therefore, the time and cost of saving and recovering the checkpointed files need to be included in the overall time and cost of the checkpointing technique which should not be greater than the time of restarting the application. For this purpose, the storage system where checkpoints are recorded needs to be not only stable and reliable but also fast.

A second critical parameter that needs to be carefully chosen when using the checkpoint-rollback recovery techniques is the checkpointing interval which defines the time between two consecutive checkpoints, *i.e.*, the frequency with which the application's states are recorded. Such a frequency also has an impact on the efficiency of the checkpoint strategy. On the one hand, the smaller the interval is, the higher the number of recorded checkpoints, leading to higher dump time. On the other hand, the longer the interval, the smaller the number of recorded checkpoints and the higher the recovery time. Thus, the ideal would be to adapt the checkpointing frequency according to the rate of failures or mean time between failures (MTBF). The closer the checkpoint frequency is to the frequency of failures, the more optimized the number of checkpoints will be. In Siavvas and Gelenbe [87], the authors propose a mathematical model to compute the optimal interval for application-level checkpoints of long-running loops. A single expression gives the interval by considering the program failure rate as well as dump and recovery times.

2.3 Replication

Replication has been applied to achieve fault tolerance in both distributed systems and databases where a client interacts with a replicated service. They are usually classified into three main different types: active, semi-active, and passive replication.

In the active replication scheme, also called the state-machine approach [74], all replicas process the requests received from the client so that their internal states are closely synchronized. Then, any replica can respond to the client requests to provide a low response time in the case of a crash. However, to ensure a strong consistency, all replicas must receive the requests in the same order and require deterministic processing which renders such a scheme quite costly.

Semi-active replication [24] extends active replication. While the actual processing of a request is performed by all replicas, one of the replicas, the *leader*, is responsible for performing the non-deterministic processing and inform the other ones called the *followers*.

With the passive replication technique, also called primary-backup [18], one of the replicas, *the primary*, receives the requests from the clients and returns responses. The other replicas, the *backups*, interact with the primary only and receive state update messages from the primary. This replication technique requires less processing power than the active ones and makes no assumption on the determinism of processing a request. However, like semi-active replication, the implementation of passive replication requires a mechanism to agree on the primary (*e.g.*, a leader election or group membership). If the primary fails, one of the backups takes over. This leads to a significantly increased response time in the case of failure which makes it unsuitable in the context of time-critical applications.

2.4 Fault Tolerant MPI

Some projects as MPI-FT [14] or MPI/FT [20] integrate fault tolerant support into the MPI standard where failures are totally masked from users and handled by the MPI library. Unfortunately, several studies point out that user transparent approaches exhibit poor efficiency on Exaflop platforms [11, 12].

The User Level Failure Mitigation (ULFM) interface [51] adopts an alternative approach offering to developers of applications a set of functions to implement fault tolerance taking into account the properties of the target application. ULFM includes functionalities for task failure detection and communication reconfiguration but it does not provide a strategy for data restoration.

According to Ansel *et al.* [5], the checkpointing and rollback recovery of MPI applications are typically made by using user-Level MPI libraries for checkpointing, which demands that all communications between tasks are made exclusively through MPI. Scalable Checkpoint/Restart for MPI (SCR) is probably of the most popular libraries for MPI applications[53]. It has been in production since 2007 and has several advantages over other solutions: it is a multilevel library that includes several strategies to reduce the load of critical shared resources such as the parallel file system. Another popular library, called the Distributed MultiThreaded CheckPointing (DMTCP) [5], is also an example of a user-level library for MPI applications and has been successfully used to checkpoint MPI applications running in a cloud environment [8].

MPI applications can also be checkpointed at the application level. For instance, CRAFT [85], an open source library, offer basic functionalities for the implementation of application-level checkpoints to MPI applications. According to the authors, the main advantage of such an approach is the reduction of the checkpointing overhead. Besides that, CRAFT also supports SCR, which enables checkpoint storage and recovery at the node level.

At the system level, the Berkeley Lab Checkpoint/Restart library [37] (BLCR) is probably one of the most widely used checkpoint-restart implementations. BLCR was developed initially for Linux clusters, but Azeem and Helal [8] showed that BLCR can be used to save MPI applications running in multiple EC2 VMs.

2.5 Fault tolerance in HPC applications

Efficient fault tolerance mechanism for HPC applications should consider performance and scalability issues. In an HPC context, fault tolerance mechanisms have conflicting goals as they should provide good performance in both failure-free execution and recovery while limiting the amount of resources used. As the failure rate increases proportionally to the number of nodes, large HPC applications require a high checkpointing frequency to limit the impact of rollbacks in the response time. On the other hand, frequency increasing has a direct impact in failure-free execution performance. Message logging can avoid rolling back all the tasks but at the cost of saving messages in the node memory, while the memory size per CPU available tends be smaller as the number of nodes increases. Coordinated checkpointing, does not require saving any messages but if a failure occurs all tasks need to rollback to the last checkpoint. Some hybrid protocols, combining coordinated checkpointing and message logging, have been proposed for fault tolerance of HPC application at large scale [56, 15].

Note that HPC applications that use MPI [46] can be fault tolerant when using one of the libraries or approaches discussed in Section 2.4.

3 Fault Tolerance in Clouds

This section discusses the implementation of *failure detectors* (Section 3.1), *checkpoint-rollback* (Section 3.2), and *replication* (Section 3.4) in the context of cloud environment. As presented in Section 2, they are extensively used in distributed fault tolerant systems. They have been extended to clouds by considering at which level they should be applied (application or provider) and cloud features such as elasticity, network dynamics, storage, and monetary cost.

In section 3.3, we present some of the existing storage services in public clouds, and how they can be used alongside the VMs to implement the checkpointing approach.

Finally, in Section 3.5, we discuss some existing solutions of the literature that tolerate the revocation of spot and preemptible VMs allocated by applications, guaranteeing that the latter execute correctly.

We point out that the addition of a fault tolerance feature can increase the users' final monetary cost, either because of the contract and use of storage services, extra VMs, or increment in the execution time caused by additional overheads. Thus, a critical challenge is to define which resources should be used to implement the fault tolerance feature, leading to a good trade-off between the monetary cost and reliability.

3.1 Failure Detectors in Clouds

As highlighted by Bui *et al.*[19], failure detectors (FD) in the context of clouds have to cope with several features of the environment such as elasticity, multi-purposed user services that continuously cause changes in the system, and the large scale number of nodes which makes difficult the collection of failure detection information.

Clients and providers can respectively detect faults of application tasks and physical resources. On the other hand, both of them can detect faults of virtual machines. The FD associated with an application should monitor the tasks and/or VMs state during their lifetime. In case of a VM failure, the application requires the allocation of a new VM to the provider and then restarts the tasks that were running in the failed VM in the former [92].

Some works in the literature propose the implementation of failure detection in clouds [49, 60, 99].

Xiong *et al.* [99] state that a FD for cloud environments should automatically adjust its parameters according to the dynamics of the network, which can greatly vary over time. Hence, they present the SFD, a self-tuning FD for cloud computing networks. Every SFD module has a sliding window which maintains the most recent samples of the arrival times and, at the next timeout delay, the parameters are adjusted using both the information in the sliding window and information from the FD output meeting, therefore, recent network conditions.

The adaptive failure detector AFD for cloud computing infrastructure [60] exploits autonomic techniques and does not rely on failure history. It continuously monitors the cloud execution and collects runtime performance data and then extracts the most relevant metrics, used to detect possible failures. When the latter is verified, the AFD adapts itself to these new detection results.

Since clouds are composed of several non-overlapping layers (*e.g.*, IaaS, SaaS, and PaaS), Lee *et al.* [49] argue that having a single heartbeat-based FD is not a good solution as failures should be distinguished. For instance, failures in the system from those of the application or from power supply. Thus, they propose to group cloud environment components into linear dependent layers. Based on such layers, their FD solution can determine the faulty layer without needing to conduct fault detection in all layers.

3.2 Implementing Checkpoints in Cloud

A checkpoint can be implemented into one of the three following distinct levels, according to the degree of transparency and location in the software stack [8, 40, 87]: i) application-level, ii) user-level, and iii) system-level.

At the application-level, the code of the application needs to inform when checkpoints should be taken. Then, the checkpoint procedure captures the state of the application through direct interaction with it. Such an approach is expected to be the most efficient one since the programmer knows which data structures and variables must be preserved and which may be discarded [72, 87]. However, its applicability is restricted to the case where the application's code is available to be modified. Furthermore, the recovery time might be a concern as it consists of the time to request, boot up, and configure a new VM to charge the application.

At the user-level, checkpoints are implemented in the user space and provide transparency to the application by virtualizing system calls. According to [40], such a virtualization allows checkpoint tools to capture the state of the entire process without being tied to the kernel, providing thus more portability between platforms, but at the cost of a constant virtualization overhead. Moreover, user-level checkpoints are usually larger than application-level ones since they cannot take advantage of memory optimization based on application semantics.

System-level checkpoint procedures are implemented either in the kernel or as a kernel module. In this case, the whole memory stack of the application is saved. Different from the user-level implementation, checkpointing at the system level does not need to virtualize system call interfaces since it has direct access to the kernel structures [40]. However, they are often linked to the kernel version making them not portable between different platforms.

3.2.1 Bag-of-Tasks applications

Bag-of-Tasks applications are composed of independent jobs (or tasks) which can thus be executed in parallel in any order. Such lack of task dependency simplifies the checkpointing implementation since the latter does not require coordination between tasks. In other words, each task can take its checkpoint independently.

CRIU [31], a very popular checkpointing tool, has been used in several works to guarantee reliability for applications running in clouds [42, 93]. It runs at the user-level and saves the full state of the process without any changes in the application code. In Teylo *et al.* [94], the authors applied CRIU to record the checkpoints of BoT applications running in a Amazon EC2 cloud.

Note that defining a good failure rate in cloud environments is not a straightforward task, particularly on the client-side. Consequently, in clouds, the checkpoint intervals are typically either user-defined fixed intervals or adaptive ones [4].

3.3 Reliable Cloud Storage Solutions

Cloud providers have several storage services available to rent. Until January 2022, Amazon Web Services (AWS) offers a total of 11 storage services divided into seven categories [83], while Google Cloud Provider (GCP) offers nine different storage services divided into eight categories [63]. Each storage service focuses on different needs in an institution's workflow. For example, AWS DataSync [77] and GCP's Data Transfer Services [70] have a major concern on data migration while Amazon Simple Storage Service (S3) [76] and GCP's Cloud storage [61] on storing data in form of objects, without an underlying file system.

Files resulting from checkpoints can be of huge sizes. Furthermore, they need to be saved fast and easily retrieved. The general storage services based on an object, file, or block storage, give all guarantees in terms of reliability and are usually cheaper than specialized storage services. Thus, in this section, we discuss these services in both AWS and GCP.

Amazon S3 and Cloud Storage are the object storage services from AWS and GCP, respectively. They similarly represent the objects, using a two-level organization [61, 76]. At the superior level, they use buckets, which are structures similar to folders that have a unique global name and help the organization of data from different users, identifying and billing them accordingly. In S3, each bucket is restricted to a single region, and each account can associate up to 100 buckets. In Cloud Storage, the user can configure the bucket availability to a single cloud region, in two close regions (dual-region), or several regions spread in a larger area (multi-region). There is no limit in GCP associated with the number of buckets in a single account, but there are bounds regarding the bucket's name and creation rate [62].

Objects are the inferior level of these two storage services. They contain the user stored data represented by a name and unique key used to access the object[1,2]. Both services have an upper limit to a single object size of 5TB [62, 76] and allow the user to create, change, and read objects from a bucket using a single operation. However, if the user wants to rename or move the object to another place, it takes at least two operations, downloading the object to a local system and then uploading it with the new name or to the new location.

The block storage service of AWS is called Amazon Elastic Block Service (EBS) [78], and the ones of GCP are Persistent Disk [68] and Local SSD [66]. In these services, the user creates storage volumes and attaches them to directories inside a Virtual Machine (VM) of each provider [73]. EBS allows the user to allocate disks from 1 GB to 156 TB [78], and these volumes can be Solid State Drives (SSDs), with low latency, or Hard Disk Drives (HDDs), with higher throughput. AWS restricts the availability of an EBS volume to a single zone (data center) of a region. In the Persistent Disk service of GCP, it is possible to create HDDs or SSDs volumes in a single cloud zone or all zones of a cloud region. The size limit for the former is 10 GB to 64 TB and for the latter is 200 GB to 64 TB [69].

Both AWS and GCP allow the user only to increase the size of the volume while attached to a VM[3,4]. Besides, a specific type of AWS's EBS volume and all GCP's Persistent Disk types can be attached to multiple VMs in read-only mode. However, the user does not know the exact physical location of an EBS or Persistent Disk volume. GCP's Local SSD service allows users to physically attach an SSD volume of size 375 GB to a single instance, offering higher performance and lower latency compared to GCP's Persistent Disks [66].

Regarding the file storage services, AWS offers EFS [80] and FSx [82], while GCP offers only Filestore [64]. FSx focuses on application migration from on-premise clusters to cloud resources. The user can choose from four high-performance file systems (NetApp ONTAP, OpenZFS, Windows File Server, and Lustre), making it easier to connect FSx to a local machine and send data to AWS. On the other hand, Amazon EFS and GCP Filestore provide a simple and scalable file system.

[1] https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingObjects.html, last access in July 19th, 2022

[2] https://cloud.google.com/storage/docs/naming-objects, last access in July 19th, 2022

[3] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-modify-volume.html, last access in July 19th, 2022

[4] https://cloud.google.com/compute/docs/disks/working-with-persistent-disks#resize_pd, last access in July 19th, 2022

They increase and decrease their allocated size automatically when the user adds or removes files. Both are compatible only with the Network File System (NFS).

The main advantages of these three file storage systems are their availability in all zones of a single region and the accessibility in parallel by several VMs, up to 500 VMs in GCP [65] and 120 VMs in AWS [81]. However, GCP imposes a 16 TiB size limit on a single file, and AWS establishes a 47.9 TiB size limit on a single file.

3.3.1 Choice of the storage service

In case of using checkpoint to tolerate failures, the choice of the most suitable storage service mainly depends on the checkpointing patterns of the application. If the implementation requires a dedicated space to store the checkpoints with a single task accessing it per time, the straightforward choice is using a local volume from Amazon EBS or GCP's Local SSD. However, in this case, the price depends not only on what it is stored but also the size of the volume. On the other hand, if the application stores concurrent multiple checkpoints or different tasks access one's checkpoint simultaneously, it is possible to use the object storage services (Amazon S3 or GCP's Cloud Storage) or the file storage systems (Amazon EFS or GCP's Firestore). The main difference between them is that the object storage services are cheaper than the file storage ones while the latter usually have better performance than the former.

In Teylo *et al.* [95], the authors compared the dump and recovery time of a checkpoint stored in Amazon S3, Amazon EBS, and Amazon EFS. They showed that the fastest service to store the checkpoint is the EBS, while the EFS is the fastest in its recovery (Figures 1 and 4 in Teylo *et al.*[95]). Moreover, the time to store sequential checkpoints in Amazon S3 consists of 37.1% of the total execution time, while in EBS corresponds only to 11.4% (Figure 3 in Teylo *et al.*[95]). However, most works in the literature use Amazon S3 as the storage service for storing checkpoints [93, 101, 102] and very few ones use a spare EBS [90]. Such a difference can be explained due to the possibility of concurrent access to a bucket while each volume of EBS is limited to a single VM at a time which in several types of HPC applications is not viable. An example of such applications is those with several independent tasks executing in parallel (Bag-of-Tasks) with data dependencies between them (workflows) or data exchange. On the other hand, in Amazon, it is more costly to store multiple checkpoints in EFS than in S3. For example, storing 30GB for a month in S3 costs $21.91 while in EFS costs $30.19 [95].

According to Nicolae and Cappello [58], one strong argument for using Amazon EBS instead of Amazon S3 to store checkpoints is that most of the time the VMs uses only part of the attached volume instead of the full allocated size, leaving a huge portion of its storage to be paid without use. Therefore, the authors propose a shared pool, composed of all spare disk spaces, to store and recover the checkpoints of applications. As several disks are used, different parts of the pool can be accessed simultaneously. To further benefit from this multitude of disks, all checkpoints are divided into small pieces and distributed among the disks so that they can be recovered in smaller time.

3.4 Replication

Depending on the level of control on the placement of each replicated task, we can divide replication in clouds based on either the provider or the client's views. The former sees the virtual machines (VMs) apart from the physical ones, which allows the deployment of different replicas into different physical resources. On the other hand, a client cannot dictate where to deploy their VMs. In 2017, AWS released the spread placement group approach that allows users to request the placement of their VMs in distinct hardware[5] but limited to seven VMs per cloud zone [84]. Thus, if the user needs more than seven VMs for her/his application, the only way to guarantee the mapping to different resources is by choosing a different cloud zone per placement group (seven VMs). However, in this case, communication time between the deployed VMs considerably increases as well as their data access time, which can become prohibitive to some applications. Due to such performance issues, most works found in the literature assume that different VMs are in separated physical resources, even in the same cloud region. It is also worth pointing out that the task replication approach increases the total execution cost for the client since he/she pays for the execution of every replica. This higher monetary cost justifies why there exist more fault-tolerant solutions in clouds based on checkpointing than on replication because the former does not consume more resources than the strictly necessary [58].

To the best of our knowledge, there is only one work concerning replication in the provider's view. Qiu *et al.* [71] presented an active task replication framework that executes the clients' jobs, each job mapped as a set of virtual machines with distinct tasks. The framework focuses on increasing reliability and performance and decreasing energy consumption. After receiving a client request, the framework actively creates replicas for each VM and allocates them to different and heterogeneous resources.

There are some works on task replication from the client's view, in which the physical data center of each VM is unknown. In [104], Zhu *et al.* present a passive replication technique, in which all tasks of a workflow have a primary and a backup copy. They schedule them in different VMs and balance the number of primary copies between all deployed instances. Li *et al.* [50] and Xie *et al.* [98] propose task replication in clouds with a variable number of replicas per task. Both papers use empirical fault rates in a Poisson distribution to estimate the number of copies per task, aiming at minimizing costs. Li *et al.* consider a deadline constraint and thus need to schedule all replicas while Xie *et al.* consider reliability bound, which allows the removal of those duplicates that surpass such a limit in order to reduce costs.

[5] https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-spread-placement-groups-for-amazon-ec2/, last access August 1st, 2022

Consequently, the latter ensures lower reliability than the former but with lower execution costs.

Another interesting paper is the one by Nik *et al.* [55] where the authors propose an active task replication solution that does not increase execution costs. Tasks are replicated in the idle slots of the scheduling solution and there exists a maximum of one replica per task. As the idle spaces may not correspond to every task in the job, it is possible to calculate the probability of each task failure in the primary assigned virtual machine and to use this probability together with the expected execution time of each task to select the ones to replicate. However, the approach does not guarantee a minimum of reliability to all tasks since some of them will not have replicas.

3.5 Fault Tolerance and Preemptible VMs

Preemptible VMs are offered with a steep discount but can be revoked at any time by the cloud provider. Therefore, fault tolerance techniques are mandatory for long-running applications using these VMs in order to ensure their complete execution. In the related literature, several works have been proposed to explore preemptible (also called spots) VMs to reduce the monetary cost of the executions. The majority of them rely on checkpoint-rollback approaches to guarantee that applications will finish even if revocations occur. Moreover, on-demand VMs are generally used as a backup resource. In this way, when a spot VM is revocated, the application is typically resumed on an on-demand VM. In Sharma *et al.*[86], for instance, the authors proposed SpotCheck, a framework that uses nested VMs within spot VMs to provide the illusion of a platform that provides always-available VMs. In order to cope with spot revocations, the nested VMs are migrated to an on-demand VM whenever a spot revocation occurs.

AutoBot [96] uses both spot and on-demand VMs for executing applications with a user-defined deadline. The framework migrates applications from spot to on-demand VMs to satisfy time constraints. It also uses checkpoint strategies to ensure reliability when executing on the preemptive VMs.Yi *et al.* propose in [101] an adaptive checkpoint that takes into account the history of the price of the spot VMs to predict their revocation and decide when a checkpoint should be recorded.

In Subramanya *et al.*[90], the authors implement a proactive mechanism, where the number of checkpoints is neither related to the VMs' volatility nor the number of revocations, but on a given checkpointing interval. In Varshney and Simmhan [96], three checkpoint strategies are proposed: i) optimistic checkpoint, where the state of the task is recorded just before the migration to an on-demand VM; ii) grace period checkpoint, where the two minutes between the notification of the interruption of a spot VM and the VM interruption itself are used to take the checkpoint; and iii) sliding checkpoint, where the checkpoint is taken in fixed intervals.

A framework that exploits both spot and on-demand VMs to execute Bag-of-Tasks applications is proposed by Teylo *et al.* [93]. It aims at minimizing the execution's monetary cost, respecting a deadline defined by the user. Periodically, the state of the application is recorded by checkpointing. Then, in case of spot VM revocation, the checkpoints are used to resume the application on on-demand VMs.

4 Conclusion and Future Directions

In this chapter, we have discussed checkpoint/rollback and replication techniques implemented in clouds and/or application tasks that provide fault tolerance to HPC applications, ensuring their correct and complete execution. However, most of the referenced solutions mainly concern applications that use only CPUs for computation. We believe that accelerators, such as GPUs and FPGAs, can be used to reduce the total execution time of different HPC applications, and, therefore, there is a growing concern around fault tolerant solutions in clouds using accelerators.

GPUs are accelerators with thousands of simple cores to execute a single instruction in multiple data while FPGAs are reconfigurable devices with logic blocks that can map different programs in specialized hardware. The increasing number of supercomputers with accelerators in the Top500 list shows that they already take part in HPC [28]. As presented by Jain and Cooperman [41], the number of clusters with Graphics Processing Units (GPUs) in the list was 136 in November 2019, growing to 151 in November 2021. At the same time, most cloud providers offer VMs with accelerators, such as GPUs and Field-Programmable Gate Array (FPGAs), to the user [67, 79]. However, to the best of our knowledge, there are few (resp., any) works in the literature that propose checkpoints on GPUs (resp. FPGAs) on Clouds.

Since GPU architectures evolve in a constant and fast way, any effort to create a generic checkpoint solution is very difficult. Therefore, most existing ones become very fast obsolete [33, 43, 59, 91]. Jain and Cooperman [41] present a GPU checkpoint approach in an initial development stage, suitable only for small applications. Hence, HPC applications using GPUs on Clouds require application-level checkpoints or some other fault-tolerant techniques to ensure their correct and complete execution in case of failures.

Lee and Son [48] propose both application-level checkpoint and live migration to reduce computational costs when training Deep Learning tasks on Clouds. The model weights are saved after each training epoch, used thus as checkpoints. Furthermore, the spot VMs price per region is monitored aiming at migrating tasks to the cheapest region. In Zhou *et al.* [103], a fault-tolerant stencil computation to AWS GPU instances, based on two-phased application-level checkpoints is presented. The first phase blocks the execution of the stencil while coping the GPU memory block to the host memory while the second one sends this memory block to a backup server asynchronously. The two-phased application-level checkpoints behave as a pipeline to surpass the communication overhead between the host and the backup server. Brum *et al.* [17] present a framework to execute a sequence alignment application on AWS spot VMs. The goal is to minimize the monetary cost, considering user-defined deadline constraints. Application-level checkpoints periodically save rows of the computed matrix in order to find the optimal sequence alignment. When a spot VM is revoked, its execution is restarted in another VM from the last saved row.

Regarding FPGA checkpointing, as it is a reconfigurable hardware, basically, two different checkpoint approaches are used: the first one is applied to the task in the FPGA; the second one concerns hardware configuration itself [44]. The former is more restricted as it needs to be restored in the same device with the same hardware configuration while in the latter, the computation can be restarted in another device.

Most early works focus on executing multiple tasks in the same FPGA to allow preemption and context switch inside a single FPGA. Therefore, they present several mechanisms to stop and restore the execution of the concurrent tasks, using only task-level FPGA checkpoints. However, when we think of a fault-tolerant context, this checkpoint approach cannot be considered a generic one due to the restriction in the restore. In Koch *et al.* [44], the authors propose the first formal model for hardware checkpoints with different mechanisms to change each hardware module, improving, thus, the capability of checkpointing. Since this first formal model, there have been several others, based on signal collection to reconstruct FPGAs checkpointing execution traces [34, 38, 39, 88]. On the other hand, all these models need particular hardware, which increases overhead costs to the FPGA synthesis, rendering them unpractical in most cases [29, 36].

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On Implementing Omega with Weak Reliability and Synchrony Assumptions. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC '03, p. 306–314. Association for Computing Machinery, New York, NY, USA (2003)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, p. 328–337. Association for Computing Machinery, New York, NY, USA (2004)
3. Alvisi, L., Marzullo, K.: Message logging: pessimistic, optimistic, causal, and optimal. IEEE Transactions on Software Engineering **24**(2), 149–159 (1998)
4. Amoon, M., El-Bahnasawy, N., Sadi, S., Wagdi, M.: On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems. Journal of Ambient Intelligence and Humanized Computing **10**(11), 4567–4577 (2019)
5. Ansel, J., Arya, K., Cooperman, G.: DMTCP: Transparent checkpointing for cluster computations and the desktop. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–12 (2009)
6. Arantes, L., Greve, F., Sens, P., Simon, V.: Eventual Leader Election in Evolving Mobile Networks. In: Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304, OPODIS 2013, p. 23–37. Springer-Verlag, Berlin, Heidelberg (2013)
7. Arévalo, S., Anta, A.F., Imbs, D., Jiménez, E., Raynal, M.: Failure Detectors in Homonymous Distributed Systems (with an Application to Consensus). J. Parallel Distrib. Comput. **83**(C), 83–95 (2015)
8. Azeem, B.A., Helal, M.: Performance evaluation of checkpoint/restart techniques: For MPI applications on Amazon cloud. In: 2014 9th International Conference on Informatics and Systems, pp. PDC–49. IEEE (2014)

9. Bertier, M., Marin, O., Sens, P.: Performance analysis of a hierarchical failure detector. In: International Conference on Dependable Systems and Networks, 2003 (DSN), pp. 635–644 (2003)
10. Bonnet, F., Raynal, M.: Anonymous asynchronous systems: the case of failure detectors. Distributed Comput. **26**(3), 141–158 (2013)
11. Bosilca, G., Bouteiller, A., Brunet, E., Cappello, F., Dongarra, J.J., Guermouche, A., Hérault, T., Robert, Y., Vivien, F., Zaidouni, D.: Unified model for assessing checkpointing protocols at extreme-scale. Concurr. Comput. Pract. Exp. **26**(17), 2772–2791 (2014)
12. Bougeret, M., Casanova, H., Robert, Y., Vivien, F., Zaidouni, D.: Using group replication for resilience on exascale systems. Int. J. High Perform. Comput. Appl. **28**(2), 210–224 (2014)
13. Bouteiller, A., Bosilca, G., Dongarra, J.J.: Redesigning the message logging model for high performance. Concurr. Comput. Pract. Exp. **22**(16), 2196–2211 (2010)
14. Bouteiller, A., Bosilca, G., Dongarra, J.J.: Redesigning the message logging model for high performance. Concurr. Comput. Pract. Exp. **22**(16), 2196–2211 (2010)
15. Bouteiller, A., Hérault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols for many-core clusters. Concurr. Comput. Pract. Exp. **25**(4), 572–585 (2013)
16. Bouteiller, A., Ropars, T., Bosilca, G., Morin, C., Dongarra, J.J.: Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In: Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA, pp. 1–9. IEEE Computer Society (2009)
17. Brum, R.C., Sousa, W.P., Melo, A.C.M.A., Bentes, C., de Castro, M.C.S., Drummond, L.M.A.: A Fault Tolerant and Deadline Constrained Sequence Alignment Application on Cloud-Based Spot GPU Instances. In: L. Sousa, N. Roma, P. Tomás (eds.) Euro-Par 2021: Parallel Processing, pp. 317–333. Springer International Publishing, Cham (2021)
18. Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: The Primary-Backup Approach, p. 199–216. ACM Press/Addison-Wesley Publishing Co., USA (1993)
19. Bui, K.T., Vo, L.V., Nguyen, C.M., Pham, T.V., Tran, H.C.: A fault detection and diagnosis approach for multi-tier application in cloud computing. J. Commun. Networks **22**(5), 399–414 (2020)
20. Buntinas, D., Coti, C., Hérault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. Future Gener. Comput. Syst. **24**(1), 73–84 (2008)
21. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. J. ACM **43**(2), 225–267 (1996)
22. Chandy, K.M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
23. Chen, W., Toueg, S., Aguilera, M.K.: On the Quality of Service of Failure Detectors. IEEE Trans. Comput. **51**(1), 13–32 (2002)
24. Chereque, M., Powell, D., Reynier, P., Richier, J.L., Voiron, J.: Active replication in Delta-4. In: [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing, pp. 28–37 (1992)
25. D'Antoni, J.: The Night the Lights Went Out in the Cloud: Lessons from the AWS Outage. https://redmondmag.com/articles/2020/12/02/lessons-from-aws-outage.aspx. Accessed: 2022-03-20
26. Das, A., Gupta, I., Motivala, A.: SWIM: scalable weakly-consistent infection-style process group membership protocol. In: Proceedings International Conference on Dependable Systems and Networks (DSN), pp. 303–312 (2002)
27. Dichev, K., Sensi, D.D., Nikolopoulos, D.S., Cameron, K.W., Spence, I.: Power Log'n'Roll: Power-Efficient Localized Rollback for MPI Applications Using Message Logging Protocols. IEEE Transactions on Parallel & Distributed Systems **33**(06), 1276–1288 (2022)
28. Dongarra, J., Luszczek, P.: TOP500, pp. 2055–2057. Springer US, Boston, MA (2011)
29. Egwutuoha, I.P., Levy, D., Selic, B., Chen, S.: A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. The Journal of Supercomputing **65**(3), 1302–1326 (2013)

30. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
31. Emelyanov, P.: Criu: Checkpoint/restore in userspace, july 2011. https://criu.org (2011)
32. García, Á.L., del Castillo, E.F., Plasencia, I.C.: An efficient cloud scheduler design supporting preemptible instances. Future Generation Computer Systems **95**, 68–78 (2019)
33. Garg, R., Mohan, A., Sullivan, M., Cooperman, G.: CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 302–313 (2018)
34. Goeders, J., Wilton, S.J.E.: Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(1), 83–96 (2017)
35. Gómez-Calzado, C., Lafuente, A., Larrea, M., Raynal, M.: Fault-Tolerant Leader Election in Mobile Dynamic Distributed Systems. In: IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 78–87 (2013)
36. Hale, R., Hutchings, B.: Enabling Low Impact, Rapid Debug for Highly Utilized FPGA Designs. In: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pp. 81–813 (2018)
37. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (blcr) for linux clusters. In: Journal of Physics: Conference Series, vol. 46, p. 067. IOP Publishing (2006)
38. Holanda Noronha, D., Zhao, R., Goeders, J., Luk, W., Wilton, S.J.: On-Chip FPGA Debug Instrumentation for Machine Learning Applications. In: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '19, p. 110–115. Association for Computing Machinery, New York, NY, USA (2019)
39. Hung, E., Wilton, S.J.E.: Scalable Signal Selection for Post-Silicon Debug. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **21**(6), 1103–1115 (2013)
40. Hursey, J.: Coordinated checkpoint/restart process fault tolerance for MPI applications on HPC systems. Indiana University (2010)
41. Jain, T., Cooperman, G.: CRAC: Checkpoint-Restart Architecture for CUDA with Streams and UVM. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2020)
42. Jesus Leonardo; Drummond, L.M.A., Oliveira, D.d.: Eeny meeny miny moe: Choosing the fault tolerance technique for my cloud workflow. In: Latin American High Performance Computing Conference, pp. 321–336. Springer (2017)
43. Jiang, H., Zhang, Y., Jennes, J., Li, K.C.: A Checkpoint/Restart Scheme for CUDA Programs with Complex Computation States. International Journal of Networked and Distributed Computing **1**, 196–212 (2013)
44. Koch, D., Haubelt, C., Teich, J.: Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays, FPGA '07, p. 188–196. Association for Computing Machinery, New York, NY, USA (2007)
45. Koo, R., Toueg, S.: Checkpointing and Rollback-Recovery for Distributed Systems. IEEE Transactions on Software Engineering **SE-13**(1), 23–31 (1987)
46. Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., Sultana, N.: A large-scale study of mpi usage in open-source hpc applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3295500.3356176. URL https://doi.org/10.1145/3295500.3356176
47. Larrea, M., Anta, A.F., Arévalo, S.: Implementing the weakest failure detector for solving the consensus problem. Int. J. Parallel Emergent Distributed Syst. **28**(6), 537–555 (2013)
48. Lee, K., Son, M.: DeepSpotCloud: Leveraging Cross-Region GPU Spot Instances for Deep Learning. In: 2017 IEEE 10th Int. Conf. on Cloud Computing (CLOUD), pp. 98–105 (2017)
49. Lee, Y.L., Liang, D., Wang, W.J.: Optimal Online Liveness Fault Detection for Multilayer Cloud Computing Systems. IEEE Transactions on Dependable and Secure Computing (2021)

50. Li, Z., Yu, J., Hu, H., Chen, J., Hu, H., Ge, J., Chang, V.: Fault-tolerant scheduling for scientific workflow with task replication method in cloud. In: V. Munoz, R. Walters, F. Firouzi, G. Wills, V. Chang (eds.) IoTBDS 2018 - Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security, pp. 95–104. SciTePress (2018)
51. Losada, N., González, P., Martín, M.J., Bosilca, G., Bouteiller, A., Teranishi, K.: Fault tolerance of MPI applications in exascale systems: The ULFM solution. Future Gener. Comput. Syst. **106**, 467–481 (2020)
52. Manvi, S.S., Shyam, G.K.: Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. Journal of network and computer applications **41**, 424–440 (2014)
53. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2010)
54. Mostefaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: International Conference on Dependable Systems and Networks (DSN), pp. 351–360 (2003)
55. Mousavi Nik, S.S., Naghibzadeh, M., Sedaghat, Y.: Task replication to improve the reliability of running workflows on the cloud. Cluster Computing **24**(1), 343–359 (2021)
56. Ndiaye, N.M., Sens, P., Thiare, O.: Performance comparison of hierarchical checkpoint protocols grid computing. Int. J. Interact. Multim. Artif. Intell. **1**(5), 46–53 (2012)
57. Newton, C.: How a typo took down S3, the backbone of the internet. https://www.theverge.com/2017/3/2/14792442/amazon-s3-outage-cause-typo-internet-server. Accessed: 2022-03-20
58. Nicolae, B., Cappello, F.: BlobCR: Efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots. In: SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12. IEEE (2011)
59. Nukada, A., Takizawa, H., Matsuoka, S.: NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 104–113 (2011)
60. Pannu, H.S., Liu, J., Guan, Q., Fu, S.: AFD: Adaptive failure detection system for cloud computing infrastructures. In: 31st IEEE International Performance Computing and Communications Conference, IPCCC 2012, Austin, TX, USA, December 1-3, 2012, pp. 71–80. IEEE Computer Society (2012)
61. Provider, G.C.: Cloud Storage. https://cloud.google.com/storage (2021). Accessed 19 December 2021
62. Provider, G.C.: Quotas & limits - Cloud Storage. https://cloud.google.com/storage/quotas (2021). Accessed 19 December 2021
63. Provider, G.C.: Cloud Computing Services. https://cloud.google.com/products/storage (2022). Accessed 11 January 2022
64. Provider, G.C.: Filestore. https://cloud.google.com/filestore (2022). Accessed 11 January 2022
65. Provider, G.C.: Limits - Filestore. https://cloud.google.com/filestore/docs/limits (2022). Accessed 12 January 2022
66. Provider, G.C.: Local SSD. https://cloud.google.com/local-ssd (2022). Accessed 11 January 2022
67. Provider, G.C.: Machine Families - Documentation. https://cloud.google.com/compute/docs/machine-types#predefined_machine_types (2022). Accessed 14 March 2022
68. Provider, G.C.: Persistent Disk. https://cloud.google.com/persistent-disk (2022). Accessed 11 January 2022
69. Provider, G.C.: Storage Options - Compute Engine. https://cloud.google.com/compute/docs/disks (2022). Accessed 11 January 2022

70. Provider, G.C.: Storage Transfer Service. https://cloud.google.com/storage-transfer-service (2022). Accessed 11 January 2022
71. Qiu, X., Sun, P., Dai, Y.: Optimal task replication considering reliability, performance, and energy consumption for parallel computing in cloud systems. Reliability Engineering & System Safety **215**, 107834 (2021)
72. Roman, E.: A survey of checkpoint/restart implementations. In: Lawrence Berkeley National Laboratory, Tech. Citeseer (2002)
73. Ruiz-Alvarez, A., Humphrey, M.: An Automated Approach to Cloud Storage Service Selection. In: Proceedings of the 2nd International Workshop on Scientific Cloud Computing, ScienceCloud '11, p. 39–48. Association for Computing Machinery, New York, NY, USA (2011)
74. Schneider, F.B.: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
75. Sens, P., Folliot, B.: Performance Evaluation of Fault Tolerance for Parallel Applications in Networked Environments. In: 1997 International Conference on Parallel Processing (ICPP '97), August 11-15, 1997, Bloomington, IL, USA, Proceedings, pp. 334–341. IEEE Computer Society (1997)
76. Services, A.W.: Amazon S3. https://aws.amazon.com/s3/ (2021). Accessed 19 December 2021
77. Services, A.W.: Amazon DataSync. https://aws.amazon.com/datasync/ (2022). Accessed 11 January 2022
78. Services, A.W.: Amazon EBS. https://aws.amazon.com/ebs (2022). Accessed 11 January 2022
79. Services, A.W.: Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/ (2022). Accessed 14 March 2022
80. Services, A.W.: Amazon EFS. https://aws.amazon.com/efs/ (2022). Accessed 11 January 2022
81. Services, A.W.: Amazon EFS quotas and limits. https://docs.aws.amazon.com/efs/latest/ug/limits.html (2022). Accessed 12 January 2022
82. Services, A.W.: Amazon FSx. https://aws.amazon.com/fsx/ (2022). Accessed 11 January 2022
83. Services, A.W.: Cloud Storage on AWS. https://aws.amazon.com/products/storage/ (2022). Accessed 11 January 2022
84. Services, A.W.: Placement Groups - Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html (2022). Accessed 1 August 2022
85. Shahzad, F., Thies, J., Kreutzer, M., Zeiser, T., Hager, G., Wellein, G.: CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. IEEE Transactions on Parallel and Distributed Systems **30**(3), 501–514 (2018)
86. Sharma, P., Lee, S., Guo, T., Irwin, D.E., Shenoy, P.J.: SpotCheck: designing a derivative IaaS cloud on the spot market. In: Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015, pp. 16:1–16:15 (2015)
87. Siavvas, M., Gelenbe, E.: Optimum interval for application-level checkpoints. In: 2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), pp. 145–150. IEEE (2019)
88. Sidler, D., Eguro, K.: Debugging framework for FPGA-based soft processors. In: 2016 International Conference on Field-Programmable Technology (FPT), pp. 165–168 (2016)
89. Strom, R., Yemini, S.: Optimistic Recovery in Distributed Systems. ACM Trans. Comput. Syst. **3**(3), 204–226 (1985)
90. Subramanya, S., Guo, T., Sharma, P., Irwin, D.E., Shenoy, P.J.: SpotOn: a batch computing service for the spot market. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015, pp. 329–341 (2015)

91. Takizawa, H., Sato, K., Komatsu, K., Kobayashi, H.: CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In: 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 408–413 (2009)
92. Tchana, A., Broto, L., Hagimont, D.: Fault tolerant approaches in cloud computing infrastructures. In: The Eighth International Conference on Autonomic and Autonomous Systems, pp. 42–48 (2012)
93. Teylo, L., Arantes, L., Sens, P., Drummond, L.M.A.: A dynamic task scheduler tolerant to multiple hibernations in cloud environments. Cluster Computing **24**(2), 1051–1073 (2021)
94. Teylo, L., Arantes, L., Sens, P., Drummond, L.M.A.: Scheduling Bag-of-Tasks in Clouds using Spot and Burstable Virtual Machines. IEEE Transactions on Cloud Computing pp. 1–1 (2021)
95. Teylo, L., Brum, R.C., Arantes, L., Sens, P., Drummond, L.M.A.: Developing Checkpointing and Recovery Procedures with the Storage Services of Amazon Web Services. In: 49th International Conference on Parallel Processing - ICPP: Workshops, ICPP Workshops '20. Association for Computing Machinery, New York, NY, USA (2020)
96. Varshney, P., Simmhan, Y.: AutoBoT: Resilient and Cost-Effective Scheduling of a Bag of Tasks on Spot VMs. IEEE Trans. Parallel Distrib. Syst. **30**(7), 1512–1527 (2019)
97. Vishwanath, K.V., Nagappan, N.: Characterizing cloud computing hardware reliability. In: Proceedings of the 1st ACM symposium on Cloud computing, pp. 193–204 (2010)
98. Xie, G., Zeng, G., Li, R., Li, K.: Quantitative Fault-Tolerance for Reliable Workflows on Heterogeneous IaaS Clouds. IEEE Transactions on Cloud Computing **8**(4), 1223–1236 (2020)
99. Xiong, N., Vasilakos, A.V., Wu, J., Yang, Y.R., Rindos, A.J., Zhou, Y., Song, W., Pan, Y.: A Self-tuning Failure Detection Scheme for Cloud Computing Service. In: 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012, pp. 668–679. IEEE Computer Society (2012)
100. Yang, R., Zhu, S., Li, Y., Gupta, I.: Medley: A Novel Distributed Failure Detector for IoT Networks. In: Proceedings of the 20th International Middleware Conference, Middleware '19, p. 319–331. Association for Computing Machinery, New York, NY, USA (2019)
101. Yi, S., Andrzejak, A., Kondo, D.: Monetary cost-aware checkpointing and migration on amazon cloud spot instances. IEEE Transactions on Services Computing **5**(4), 512–524 (2011)
102. Zhou, A.C., He, B., Liu, C.: Monetary cost optimizations for hosting workflow-as-a-service in IaaS clouds. IEEE transactions on cloud computing **4**(1), 34–48 (2015)
103. Zhou, J., Zhang, Y., Wong, W.: Fault Tolerant Stencil Computation on Cloud-Based GPU Spot Instances. IEEE Trans. on Cloud Comput. **7**(4), 1013–1024 (2019)
104. Zhu, X., Wang, J., Guo, H., Zhu, D., Yang, L.T., Liu, L.: Fault-Tolerant Scheduling for Real-Time Scientific Workflows with Elastic Resource Provisioning in Virtualized Clouds. IEEE Transactions on Parallel and Distributed Systems **27**(12), 3501–3517 (2016)