International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

An Evaluation of Major Fault Tolerance Techniques Used on High Performance Computing (HPC) Applications

Mirza Mohammed Akram Baig

Submitted: 02/11/2022 **Accepted**: 01/02/2023

Abstract: High performance computing have a high number of constituent components used to facilitate data movement. Key characteristics of these systems include parallel processing, large memory, multiprocessor or multimode communication, and parallel file systems. Though they can turnaround computing in scenarios that need maximum processing power, HPCs face many challenges, key among them being fault tolerance. Today, most applications deal with faults by noting checkpoints frequently. Whenever a fault occurs, all the processes are terminated, and the task is loaded once again from the last checkpoint. Most applications deal with faults by noting checkpoints frequently. Whenever a fault occurs, all the processes are terminated, and the task is loaded once again from the last checkpoint. Key fault tolerance techniques used on HPC applications (reactive and proactive) were evaluated in this paper. Reactive protocols discussed include checkpointing/ restarting, replication, retry, and SGuard, while proactive techniques include preemptive migration, software rejuvenation, and self-healing strategy. As seen from the discussion on the drawbacks of each approach, efficient management of faults can best be achieved by using a hybrid system applying proactive and reactive measures simultaneously.

Keywords: *management, multiprocessor, checkpoints, software rejuvenation, terminated, tolerance*

1. Introduction

High Performance Computing (HPC) includes advanced machines that utilize components similar to computers used on a daily basis. HPCs use memory, data storage, central processing units, communication, and software in a speedy manner [1]–[3]. A key feature of this type of computing is in the scale of use and the number of constituent components used to facilitate data movement. Faster communication is used in HPC applications, facilitated by highly proprietary communication schemes that attain speeds 4-100 times faster than common Ethernet [1]. These applications use parallel and simultaneous data paths to push and pull data from the central processing unit to storage and memory, and back [1]. Other terms that are used interchangeably with HPC include supercomputing and advanced computing.

The four basic characteristics of HPCs include: parallel processing, large memory, multiprocessor or multinode communication, and parallel file systems. For parallel processing, HPC applications consist of various computers with various cores in each CPU and many CPUs in a single computer. Most HPCs have CPUs ranging typically between 8 and 32 cores, but recent general purpose graphics processing units (GPUs) use hundreds of cores [4], [5]. Each supercomputer holds between 32, 64, 256, or more gigabytes of memory, and the quantity of memory depends on the science or engineering used. Multinode communication allows the use of a group of computers in one application [1], [4], [5]. Special software techniques, such as Message Passing Interface (MPI) may be used to share an application on more than one computer [6]. Lastly, the parallel file systems capability of HPCs allows the movement of data between compute nodes and storage facilities across many channels at the same time. HPCs run on an operating system – mostly Linux – and often require a machine-management software referred to as middleware[1]. Due to the peculiarities of different vendor platforms, processors, and storage disks among other options, HPCs add a lot of complexity for its users. HPCs are useful for capability or turnaround computing in situations that require the highest processing power to be utilized on one problem. However, these applications often face various challenges, key among them being fault tolerance [7], [8]. The many processors on modern computing applications mean that components such as virtual instances, communication links, integrated circuit sockets, and processors are likely to fail. A concerning issue with machines of this magnitude is the mean time between failures. Recent estimates show that a system operating with 100,000 processors is likely to report a letdown every few minutes [7], [9], [10]. A primary issue arising from this observation is how to use a machine with over 100,000 processors effectively. The main issue with HPC machines running on thousands of processors is that their peak power might be in the petaflops range,

Senior Member of Technical Staff, Illumio Inc

yet the most effective power may be a few teraflops [10]. To minimize such challenges, it is necessary to design and develop scientific algorithms that can help supercomputers efficiently use their thousands of processor machines. Fault tolerance, therefore, must ensure that HPC applications operate smoothly and simultaneously with minimal overheads and a detailed outlook of the environment.

Today, most applications deal with faults by noting checkpoints frequently. Whenever a fault strikes, all the processes are terminated, and the task is loaded once again from the last checkpoint [11], [12]. In line with this approach, this paper evaluates major reactive and proactive fault tolerance techniques used on high performance computing applications. The rest of the paper is organized as follows. Section II discusses different types of faults and failures associated with high performance computing. Key categories highlighted include hardware and software faults and failures. Section III classifies fault tolerance techniques into reactive and proactive protocols. Reactive techniques discussed include checkpointing/ restarting, replication, retry, and SGuard, while proactive include preemptive migration, software rejuvenation, and self-healing. Section IV contains the conclusion.

2. Faults And Failures

A failure refers to an event that happens when a service diverges from the rightful operation or at the bare minimum when a single outward state of the system strays from the rightful state [13], [14]. In high-performance computing, various errors, faults, or failures occur. Some are often momentary, while others are irrecoverable [8]. Some of the faults and failures cause irreversible effects immediately they strike, while others can result in the corruption of data after a very long delay. Irrecoverable faults and failures are usually the worse since they interfere with the execution of the application, a case in example being the fail-stop failures. All in all, faults mostly arise as a result of complicated interactions between internal and external factors that happen infrequently and are less likely to reproduce [8], [13].

Most scholarly studies divide faults and failures into two primary categories: software and hardware [8], [13], [14]. Each of these categories has their separate subcategories. Hardware faults and failures represent most of failures for all HPC systems. Most of the hardware failures constitute errors in memory and processor [14]. On the contrary, software errors represent close to 30% of the faults and failures in HPCs. The general principle is that as a system becomes bigger and complex, the number of software failures increases. Some contributing factors to software failures include filesystem problems, failures of the job scheduler, and challenges with the operating system [7], [13]–[16]. Nonetheless, failures with an unknown root cause can also be significant in some sites [14]. Most of the failures resulting from unidentified root causes often arise from human errors, environmental factors, or challenges with the network [14]. Overall, hardware failures are much easier to diagnose than software faults.

Hardware root causes (%)	**Software root causes (%)**	**Environmental root causes (%)**
CPU 42.8	Other software 30.0	Power Outage 48.4
Memory Dimm 21.4	OS 26.0	UPS 21.2
Node Board 6.8	Parallel File System 11.8	Power Spike 15.1
Other 5.1	Kernel software 6.0	Chillers 9.8
Power Supply 4.4	Scheduler Software 4.9	Environment 5.3
Interconnect Interface 3.1	Cluster File System 3.6	
Disk Drive 2.0	Resource Mgmt System 3.2	
Interconnect Soft Error 1.3	Network 2.7	
System Board 0.9	User code 2.4	
PCI Backplane 0.8	NFS 1.6	

Fig 1. Detailed data on the root cause of faults and failures [14].

3. Fault Tolerance Techniques

Fault tolerance in computing systems is an area that has been enriched through many years of research. Issues relating to fault tolerance have been highlighted in different areas of computing systems, including operating systems, computer architecture, mobile computing, distributed systems, and computer networks. Even though advances have been made in tolerating faults, each new area presents new challenges for which past techniques have restricted applicability [17]. This section discusses existing fault tolerance techniques which exist in two major categories: reactive and proactive fault tolerance techniques.

a. Reactive fault tolerance techniques

Reactive fault tolerance techniques minimize the impact of failures on the application once the failure has

happened [7], [18]–[21]. Examples of policies under this category include checkpointing, replication, retry, and SGuard.

i) Checkpointing/ Restart

The checkpointing/ restart technique allows the system to restart a task from the most recent check point instead of the first phase [20]. Checkpointing can be performed in coordinated and uncoordinated protocols and involves saving a snapshot image of the current state of the application [8], [22]. The snapshot image is later used for reinitiating the execution in the event of failure. Checkpointing also involves recomputing the unaccounted-for parts of the execution. Due to the capability that this technique offers, it is useful in long running and big applications. Checkpointing can be of various types, including process checkpointing, coordinated checkpointing, uncoordinated checkpointing, and hierarchical checkpointing.

One type often used in HPCs is process checkpointing [8], [23]. In most HPC applications, a process features various threads (could be at the user or system level), making the process a comparable application on its own. Process checkpointing seeks to mark and save the current state of a process within HPCs. The technique relies on coarse grain locking mechanism to fleetingly disrupt the implementation of all process threads. The interruption provides the checkpointing mechanism with the ability to have a global view of the current state and minimize the challenge of saving the state of the process to a subsequent problem. Most modern process checking protocols can depend on an operating system extension [24][1], dynamic libraries [24], [25], compilers [26], [27], user-level API [28], [29], or routines defined by the user to create an application specific checkpoint [11].

Coordinated checkpointing is another popular method used to achieve fault tolerance, more so in distributed systems. The main goal of this approach is to establish a coherent distributed view of the distributed system [8], [30]–[32]. During the period free of failures, a coordinated checkpoint protocol records the status of the application and messages that pass through the network in stable storage. Whenever a failure happens, the recovery process entails resetting the application to the last available status and reinitiating the execution process. Compared to other checkpointing methods, the coordinated technique is much simpler and the system can tolerate concurrent failures [32], [33]. Besides, the garbage collection process is much easier and efficient in coordinated checkpointing since only the last checkpoint of each process is required [34]. Though advantageous, coordinated checkpointing has certain drawbacks. First, the protocol is expensive in terms of the consumption of energy since one failure can make all the processes rollback to their previous checkpoint [34], [35]. Second, the protocol's approach of having all processes write their checkpoints concurrently creates an eruption of access to the I/0 system, and this may affect the speed of executing the system [34], [36].

Another family of checkpointing protocols is uncoordinated checkpointing. This technique does not rely on synchronization between the processes during checkpoints [34], [35]. Due to its capabilities, uncoordinated checkpointing can be used to address the issue of burst accesses to the input and output system by providing room for better scheduling of checkpoints [34], [35]. HPCs using uncoordinated checkpoints must, however, be aware of its potential drawback. If none of the set checkpoints create a coherent global state, the application would need restarting from the beginning whenever failures happen [8], [23], [34], [35], [37], [38]. This domino effect increases the cost of recovery and complicates the garbage collection process.

Hierarchical checkpointing techniques also exist alongside the above protocols. These approaches attempt to put together coordinated and rollback processes alongside uncoordinated checkpointing with message logging [8], [23], [39]. Stated otherwise, hierarchical checkpointing tries to keep the best of the two approaches. The protocol distributes processes in groups. Processes in the same group link up their checkpoints and rollbacks, while uncoordinated checkpointing is effected between groups [8], [23], [39]. This also means that the status of one process depends on the exchanges between groups, and with other processes inside the group.

ii) Replication

Apart from checkpointing, HPC systems can also cope through replication. Replication operates on the basis of reproducing all computations [8], [23], [40]. The protocol groups the processors in pairs, in a manner that allows every processor to have a replica. In this case, a replica refers to another processor carrying out similar computations and receiving analogous messages [8], [23]. When the processor is affected by a fault, the replica remains unimpacted and the implementation of the application can still go on until the replica itself is affected by a fault later on. The operational process of replication often seems expensive since half of the resources are usually wasted, along with the overhead costs involved in maintaining a reliable state between the two processors of each pair [8], [23], [41]–[43]. Another challenge with replication is the selection and placement

[1] The Berkeley Lab Checkpoint/ Restart (BLCR) operating system extension provides an entirely open checkpoint of the whole process. It is possible to restore the checkpoint on the same hardware, with the same software environment. The checkpointing application saves the whole state from CPU registers to the virtual memory map, guaranteeing that the function call stack is saved and restored without much intervention.

of the replica since the system used for storage is often large and complicated in nature. This means that replicas occupy unnecessary storage space and do not necessarily improve the operation of the system [19], [44]. While replication can be used independently, it is also possible for HPC systems to use them in combination with checkpointing protocols as demonstrated in [8], [23], [45].

Iii) Retry

Transient type of faults in HPCs can also be managed using a 'Retry' protocol [19], [46]. Upon the detection of a fault in the system, the protocol applies a retry mechanism in an attempt to recuperate from the effect of the fault. Once the retry mechanism has been activated, the defective module attempts its activity once again for a certain time period [19], [46]–[48]. In the event that the fault persists much longer than the retry period set in the system, it is considered as an irreversible fault. In such cases, a faulty node has to be replaced [19], [46]. If a fault recedes in between the retry period, it is categorized as a transient fault and the system resumes its normal functioning upon recovery. For this protocol to work effectively, the retry period must tarry long enough to allow the fault to go, as well as short enough to prevent the intersecting of faults [19], [46]–[48].

iv) SGuard

The SGuard protocol is a relatively recent rollback and recovery-based technique [19], [49], [50]. It is suitable for instantaneous video streaming and experiences much less turbulence since it is developed by a combination of reactive fault techniques, such as checkpointing, rollback, recovery and replication [49]. The approach has mostly been applied to deal with faults affecting stream processing engines (SPEs) that are installed in various clusters. SGuard initiates checkpoints asynchronously as the system is running [49], [50]. The protocol rolls back and restores the failed servers using the last well-known functional checkpoint. Checkpointing, together with the other processes such as rollback and recovery of failed servers happens asynchronously without encountering interruptions [49], [50]. SGuard saves the checkpointed states on distributed file systems (DFS), such as GFS, HDFS, or Amazon EC_2 [50]. Besides, it masks failures using the replication approach. The protocol can manage both hardware crashes and software failures and is classified as a less troublesome solution for HPCs that facilitate instantaneous video streaming [49], [50].

b. Proactive fault tolerance techniques

Proactive fault tolerance techniques are protocols used to avoid imminent failure through prediction. These measures suggest that it is possible for HPCs to anticipate failures and take proactive action before the failure happens. This is beneficial since it reduces the impact of failure and increases the chances of successful execution of applications [7], [51]–[53]. Examples of processes, which are further discussed below, include software rejuvenation, self-healing, and preemptive migration.

i) Software rejuvenation

A software is prone to aging as it runs for long periods of time and as internal errors accumulate [54]–[56]. The aging phenomenon can potentially lead to a degradation in performance and depletion of progressive resources. Eventually, these factors lead to software crash [56]. Some of the consequences of software aging include data inconsistency, exhaustion of the resources within an operating system, and numerical errors [57], [58]. As such, software aging is one of the major obstacles to achieving high availability of HPCs. The aging obstacle can, however, be countered through software rejuvenation techniques. These protocols are designed to reduce the chances of future erratic HPC application outages [54]–[56], [59].

Software rejuvenation involves two major processes. The first is to forecast a state when errors are likely to occur in a system and the amount of time that would be needed to resolve the error [54], [60], [61]. The second process is to get the software from a state prone to failure to one that is free from faults. To apply software rejuvenation protocols, the aging process must be modeled. The models would provide a decent estimate of the current or estimated progress of the failure state of a system. Models can also help plan optimal rejuvenation times or the management of system administrator tasks, such as notifying the operators in case of expected crashes [8], [23]. This is a key reason why software rejuvenation is classified as an adaptive or proactive fault tolerance technique. Examples of this approach include software restart or system reboot. While these approaches are efficient, they can lead to service downtime when the application becomes unavailable [57], [62].

During this phase, the most insightful and widely adopted approach is to cease software transiently. The protocol also cleans up the internal runtime states and restarts it [54], [63], [64]. While using this protocol, some scholars [55] have suggested a time based approach that uses symbolic algebraic tactics. At the same time, other researchers [57] have suggested pre-checking and live migration techniques which have the ability to improve the availability of the system significantly. In general, software rejuvenation protocols have been examined in the context of application replication in an attempt to minimize service outages [8], [23], [65].

ii) Preemptive migration

The preemptive migration protocol relies on a feedback-loop control mechanism [66], [67]. The protocol monitors the health of every application and starts precautionary measures when failure threats are imminent. Once a potential failure is diagnosed, the system reallocates running application parts from the

detrimental to the healthy compute nodes [67], [68]. A feedback loop develops through the progressive monitoring of the health of the application, reallocation of application parts, and reflection of the impact of the allocation on application health [67], [68].

The protocols involved in preemptive migration monitor the health of an application using hardware and software components [66]–[68]. Some of the monitoring approaches may include examining fan speeds, the temperature of the processor, and a processor's rate of utilization [67]. The health of software can be monitored by watching their progress (for instance, their input and output patterns), in the same manner as performance monitoring. Filters are used to monitor data and trends, patterns, correlations, indications of imminent failure, and potential future threads identified through online reliability analysis [18], [19], [67], [69], [70]. Apart from reliability, the feedback-loop mechanism may consider performance factors and the application together with the health of the system may be examined based on performability [67], [71].

The reallocation process removes parts of an application from a single or more nodes and discounts them from future use [19], [67], [72]. The number of nodes reduces, and migration is implemented to idle nodes, reserved spares, or those already allocated. The system administrator manually inspects the eliminated nodes before adding them back to the pool. Though useful, reliance on the feedback-loop control mechanism can present a real-time challenge. The reallocation process must be completed before the projected failure happens. If not, the application will encounter the faults and that would require reactive techniques to be initiated [18], [19], [67], [70], [73]. The types of failures covered, and the accuracy or timeliness of migration determine the quality of a feedback-loop control mechanism. Another shortcoming is that not all failures can be forecasted, and preemptive migration may not cover all types of failures. Due to this shortcoming, combining proactive and reactive fault tolerance techniques has been shown to provide efficient coverage for both certain and uncertain failures [67], [74].

[67] classifies proactive fault tolerance using preemptive migration into four groups. Type 1 is the most basic form and constitutes a monitor in each node regularly observing the health parameters of the system. The monitor notifies the resource manager after noticing faults in operational parameters. The resource manager gets ride of all parts of the application from the compute node before redistributing it, and forewarning the runtime environment [67], [75], [76]. Type 2 features a few enhancements to the basic form of proactive fault tolerance. Rather than just notifying the resource manager after detecting faults, Type 2 uses a filter on every node to organize raw sensor data [67]. Type 3 is much more advanced than Types 1 and 2, and accumulated data from all filters is processed using reliability analysis [67], [77]. Reliability analysis helps model the system and each of the running applications. Type 4 is much more enhanced, and its reliability analysis approach uses a History Database to record reliability patterns. Type 4 provides a high quality of service with improved accuracy of predicting failures.

iii) Self-healing

The self-healing strategy refers to an attempt to minimize the effects of system degradation by automatically recovering from a fault or a sequence of faults [78]–[81]. The protocol can initiate with the help of local architecture features or through the implementation of certain fault recovery procedures. Some procedures that could be applied feature periodically applied supervision tasks. When initiated correctly, a HPC system should recover from both temporary or perpetual faults [81]. This approach also suggests that HPC systems can self-heal whether or not accurate diagnosis of faults is done. However, systems that diagnose faults come with extra advantages.

[81] illustrates the functioning of a self-healing procedure. The system first evaluates its fitness against a pattern image as part of the fault diagnosis process. Fault diagnosis happens concurrently with scrubbing activity to occasionally recover from Single-Event-Upsets (SEUs) [82], [83]. Where permanent faults are detected after reconfiguration, another evolutionary run is initiated. SEU detection happens quickly since the process evaluates the pattern of an image Recoveries from temporary faults happen very quickly. Adaptation or re-evolution of the system is only necessary when new permanent faults are reported [81]. The speed of these processes yields a recovery time of less than one minute.

4. Conclusion

Fault tolerance happens often in HPCs, and various techniques have been developed to deal with them. The paper provides a foundation for major reactive and proactive fault tolerance techniques used on high performance computing applications. Reactive protocols discussed include checkpointing/ restarting, replication, retry, and SGuard, while proactive techniques include preemptive migration, software rejuvenation, and self-healing strategy. Proactive protocols are considered a much better option since they predict and avoid failure before it happens. However, proactive measures can also fail, hence the need for reactive techniques. Efficient management of faults can also be achieved by using a hybrid system applying proactive and reactive measures simultaneously.

References

[1] A. Osseyran and M. Giles, Eds., *Industrial Applications of High-Performance Computing: Best Global*

Practices. New York: Chapman and Hall/CRC, 2015. doi: 10.1201/b18322.

[2] J. Xie, Z. Chen, C. C. Douglas, W. Zhang, and Y. Chen, Eds., *High performance computing and applications: Third International Conference, HPCA 2015 Shanghai, China, July 26-30, 2015 Revised Selected Papers*. Springer, 2015.

[3] W. Zhang, W. Tong, Z. Chen, and R. Glowinski, Eds., *Current trends in high performance computing and its applications: proceedings of the International Conference on High Performance Computing and Applications, August 8-10, 2004, Shanghai, P.R. China*. Berlin; New York: Springer, 2005. Accessed: Feb. 16, 2022. [Online]. Available: http://site.ebrary.com/id/10143448

[4] M. A. Acuna and T. Aoki, "Real-time Tsunami simulation on multi-node GPU cluster," *ACMIEEE Conf. Supercomput.*, p. 2009.

[5] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 233–240. doi: 10.1109/PDP.2013.41.

[6] X. Zhang, S. E. Wong, and F. C. Lightstone, "Message passing interface and multithreading hybrid for parallel molecular docking of large databases on petascale high performance computing machines," *J. Comput. Chem.*, vol. 34, no. 11, pp. 915–927, 2013, doi: 10.1002/jcc.23214.

[7] I. P. Egwutuoha, S. Chen, D. Levy, and B. Selic, "A Fault Tolerance Framework for High Performance Computing in Cloud," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, May 2012, pp. 709–710. doi: 10.1109/CCGrid.2012.80.

[8] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*. Cham: Springer International Publishing, 2015. Accessed: Feb. 16, 2022. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-20943-2

[9] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatch Q.*, vol. 3, no. 4, Nov. 2007.

[10] A. Geist and C. Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors." Oak Ridge National Laboratory. Accessed: Feb. 16, 2022. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.8335&rep=rep1&type=pdf

[11] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–8. doi: 10.1109/IPDPS.2007.370605.

[12] S. Chakravorty and L. V. Kale, "A Fault Tolerance Protocol with Fast Fault Recovery," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–10. doi: 10.1109/IPDPS.2007.370310.

[13] A. Gainaru and F. Cappello, "Errors and Faults," in *Fault-Tolerance Techniques for High-Performance Computing*, T. Herault and Y. Robert, Eds. Cham: Springer International Publishing, 2015. Accessed: Feb. 16, 2022. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-20943-2

[14] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 337–350, Oct. 2010, doi: 10.1109/TDSC.2009.4.

[15] A. Geist and D. A. Reed, "A survey of high-performance computing scaling challenges," *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 1, pp. 104–113, Jan. 2017, doi: 10.1177/1094342015597083.

[16] C. Engelmann and T. Naughton, "Toward a Performance/Resilience Tool for Hardware/Software Co-design of High-Performance Computing Systems," in *2013 42nd International Conference on Parallel Processing*, Oct. 2013, pp. 960–969. doi: 10.1109/ICPP.2013.114.

[17] S. Chetan, A. Ranganathan, and R. Campbell, "Towards fault tolerance pervasive computing," *IEEE Technol. Soc. Mag.*, vol. 24, no. 1, pp. 38–44, 2005, doi: 10.1109/MTAS.2005.1407746.

[18] A. Bala and I. Chana, "Fault tolerance - Challenges, techniques and implementation in cloud computing," *Int. J. Comput. Sci. Issues*, vol. 9, no. 1, pp. 288–294, Jan. 2012.

[19] A. Kumar and D. Malhotra, "Study of various reactive fault tolerance techniques in cloud computing," *Int. J. Comput. Sci. Eng.*, vol. 6, no. 5, Jun. 2018, [Online]. Available: https://www.ijcseonline.org/spl_pub_paper/IJCSE-ETACIT-2K18-010.pdf

[20] P. K. Patra, H. Singh, and G. Singh, "Fault tolerance techniques and comparative implementation in cloud computing," *Int. J. Comput. Appl.*, vol. 64, no. 14, pp. 37–42, Feb. 2013.

[21] G. R. Kalanirnika and V. M. Sivagami, "Fault tolerance in cloud using reactive and proactive techniques," *Int. J. Comput. Sci. Eng. Commun.*, vol. 3, no. 3, pp. 1159–1164, 2015.

[22] G. Aupy, A. Benoit, M. E. M. Diouri, O. Gluck, and L. Lefevre, "Energy-aware checkpointing strategies," in *Fault-Tolerance Techniques for High-Performance Computing*, T. Herault and Y. Robert, Eds. Cham: Springer International Publishing, 2015. Accessed: Feb. 16, 2022. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-20943-2

[23] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*, T. Herault and Y. Robert, Eds. Cham: Springer International Publishing, 2015. Accessed: Feb. 16, 2022. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-20943-2

[24] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/ Restart (BLCR) for Linux Clusters." Ernest Orlando Larence Berkeley National Laboratory, 2006.

[25] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A Checkpoint/Restart Tool for CUDA Applications," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dec. 2009, pp. 408–413. doi: 10.1109/PDCAT.2009.78.

[26] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurr.*

Comput. Pract. Exp., vol. 22, no. 6, pp. 749–766, 2010, doi: 10.1002/cpe.1541.

[27] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs, "Compiler-assisted full checkpointing," *Softw. Pract. Exp.*, vol. 24, no. 10, pp. 871–886, 1994, doi: 10.1002/spe.4380241002.

[28] K. Sato *et al.*, "A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2014, pp. 21–30. doi: 10.1109/CCGrid.2014.24.

[29] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current practice and a direction forward in checkpoint/restart implementations for fault tolerance," in *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005, p. 8 pp.-. doi: 10.1109/IPDPS.2005.157.

[30] G. Cao and M. Singhal, "On coordinated checkpointing in distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 12, pp. 1213–1225, Dec. 1998, doi: 10.1109/71.737697.

[31] L. Wang *et al.*, "Modeling coordinated checkpointing for large-scale supercomputers," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, Jun. 2005, pp. 812–821. doi: 10.1109/DSN.2005.67.

[32] N. Neves and W. K. Fuchs, "Coordinated checkpointing without direct coordination," in *Proceedings. IEEE International Computer Performance and Dependability Symposium. IPDS'98 (Cat. No.98TB100248)*, Sep. 1998, pp. 23–31. doi: 10.1109/IPDS.1998.707706.

[33] R. E. Strom, D. F. Bacon, and S. A. Yemini, "Volatile logging in n-fault-tolerant distributed systems," in *[1988] The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, Jun. 1988, pp. 44–49. doi: 10.1109/FTCS.1988.5295.

[34] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 989–1000. doi: 10.1109/IPDPS.2011.95.

[35] E. N. (Mootaz) Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[36] R. A. Oldfield *et al.*, "Modeling the Impact of Checkpoints on Next-Generation Systems," in *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, San Diego, CA, USA, Sep. 2007, pp. 30–46. doi: https://doi.org/10.1109/MSST.2007.4367962.

[37] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. K. Fuchs, "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 5, pp. 546–554, May 1995, doi: 10.1109/71.382324.

[38] A. Mostefaoui and M. Raynal, "Efficient message logging for uncoordinated checkpointing protocols," in *Dependable Computing — EDCC-2*, Berlin, Heidelberg, 1996, pp. 353–364. doi: 10.1007/3-540-61772-8_48.

[39] H. S. Paul, A. Gupta, and R. Badrinath, "Hierarchical cordinated checkpointing protocol." Indian Institute of Technology. Accessed: Feb. 16, 2022. [Online]. Available: https://www.angelfire.com/linux/badri/papers/PDCS-hier.pdf

[40] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-Based Fault-Tolerance for Large-Scale Graph Processing," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2014, pp. 562–573. doi: 10.1109/DSN.2014.58.

[41] J. P. Walters and V. Chaudhary, "Replication-Based Fault Tolerance for MPI Applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 7, pp. 997–1010, Jul. 2009, doi: 10.1109/TPDS.2008.172.

[42] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, Apr. 1997, doi: 10.1109/2.585156.

[43] E. B. Tchernev, R. G. Mulvaney, and D. S. Phatak, "Investigating the Fault Tolerance of Neural Networks," *Neural Comput.*, vol. 17, no. 7, pp. 1646–1664, Jul. 2005, doi: 10.1162/0899766053723096.

[44] A. Rajalakshmi, D. Vijayakumar, and K. G. Srinivasagan, "An improved dynamic data replica selection and placement in cloud," in *2014 International Conference on Recent Trends in Information Technology*, Apr. 2014, pp. 1–6. doi: 10.1109/ICRTIT.2014.6996180.

[45] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P. A. Vanrolleghem, "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 2, pp. 180–190, Feb. 2009, doi: 10.1109/TPDS.2008.93.

[46] A. M. Saleh and J. H. Patel, "Transient-fault analysis for retry techniques," *IEEE Trans. Reliab.*, vol. 37, no. 3, pp. 323–330, Aug. 1988, doi: 10.1109/24.3763.

[47] J. Sosnowski, "Transient fault tolerance in digital systems," *IEEE Micro*, vol. 14, no. 1, pp. 24–35, Feb. 1994, doi: 10.1109/40.259897.

[48] Y. Huang, P. Jalote, and C. Kintala, "Two techniques for transient software error recovery," in *Hardware and Software Architectures for Fault Tolerance*, Berlin, Heidelberg, 1994, pp. 159–170. doi: 10.1007/BFb0020031.

[49] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant system processing using a distributed, replicated file system," *Proc VLDB Endow.*, vol. 1, no. 1, pp. 574–585, Aug. 2008.

[50] M. A. Mukwevho and T. Celik, "Toward a Smart Cloud: A Review of Fault-Tolerance Methods in Cloud Systems," *IEEE Trans. Serv. Comput.*, vol. 14, no. 2, pp. 589–605, Mar. 2021, doi: 10.1109/TSC.2018.2816644.

[51] G. Vallee *et al.*, "A Framework for Proactive Fault Tolerance," in *2008 Third International Conference on Availability, Reliability and Security*, Mar. 2008, pp. 659–664. doi: 10.1109/ARES.2008.171.

[52] J. Liu, S. Wang, A. Zhou, S. A. P. Kumar, F. Yang, and R. Buyya, "Using Proactive Fault-Tolerance Approach to Enhance Cloud Service Reliability," *IEEE Trans. Cloud Comput.*, vol. 6, no. 4, pp. 1191–1202, Oct. 2018, doi: 10.1109/TCC.2016.2567392.

[53] S. Chakravorty, C. L. Mendes, and L. V. Kale, "Proactive fault tolerance in large systems", [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.644.7952&rep=rep1&type=pdf

[54] J. Liu, J. Zhou, and R. Buyya, "Software Rejuvenation Based Fault Tolerance Scheme for Cloud

Applications," in *2015 IEEE 8th International Conference on Cloud Computing*, Jun. 2015, pp. 1115–1118. doi: 10.1109/CLOUD.2015.164.

[55] D. Bruneo, S. Distefano, F. Longo, A. Puliafito, and M. Scarpa, "Workload-Based Software Rejuvenation in Cloud Systems," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1072–1085, Jun. 2013, doi: 10.1109/TC.2013.30.

[56] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software Aging and Rejuvenation: Where We Are and Where We Are Going," in *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*, Nov. 2011, pp. 1–6. doi: 10.1109/WoSAR.2011.15.

[57] M. Melo, J. Araujo, R. Matos, J. Menezes, and P. Maciel, "Comparative Analysis of Migration-Based Rejuvenation Schedules on Cloud Availability," in *2013 IEEE International Conference on Systems, Man, and Cybernetics*, Oct. 2013, pp. 4110–4115. doi: 10.1109/SMC.2013.701.

[58] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*, Nov. 2008, pp. 1–6. doi: 10.1109/ISSREW.2008.5355512.

[59] R. Matias and P. J. F. Filho, "An Experimental Study on Software Aging and Rejuvenation in Web Servers," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Sep. 2006, vol. 1, pp. 189–196. doi: 10.1109/COMPSAC.2006.25.

[60] T. Thein, S.-D. Chi, and J. S. Park, "Improving Fault Tolerance by Virtualization and Software Rejuvenation," in *2008 Second Asia International Conference on Modelling Simulation (AMS)*, May 2008, pp. 855–860. doi: 10.1109/AMS.2008.75.

[61] Y. Huang, C. M. R. Kintala, L. Bernstein, and Y.-M. Wang, "Components for software fault tolerance and rejuvenation," *T Tech. J.*, vol. 75, no. 2, pp. 29–37, Mar. 1996, doi: 10.15325/ATTTJ.1996.6771126.

[62] J. Araujo, R. Matos, P. Maciel, F. Vieira, R. Matias, and K. S. Trivedi, "Software Rejuvenation in Eucalyptus Cloud Computing Infrastructure: A Method Based on Time Series Forecasting and Multiple Thresholds," in *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*, Nov. 2011, pp. 38–43. doi: 10.1109/WoSAR.2011.18.

[63] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *IEEE Trans. Dependable Secure Comput.*, vol. 2, no. 2, pp. 124–137, Apr. 2005, doi: 10.1109/TDSC.2005.15.

[64] A. Pfening, S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Optimal software rejuvenation for tolerating soft failures," *Perform. Eval.*, vol. 27–28, pp. 491–506, Oct. 1996, doi: 10.1016/S0166-5316(96)90042-5.

[65] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Checkpointing Strategies with Prediction Windows," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, Dec. 2013, pp. 1–10. doi: 10.1109/PRDC.2013.9.

[66] D. Kochhar, A. Kumar, and J. Hilda, "An approach for fault tolerance in cloud computing using machine learning technique," *Int. J. Pure Appl. Math.*, vol. 117, no. 22, pp. 345–351, 2017.

[67] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb. 2009, pp. 252–257. doi: 10.1109/PDP.2009.31.

[68] S. Prathiba and S. Sowvarnica, "Survey of failures and fault tolerance in cloud," in *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, Feb. 2017, pp. 169–172. doi: 10.1109/ICCCT2.2017.7972271.

[69] A. Polze, P. Tröger, and F. Salfner, "Timely Virtual Machine Migration for Pro-active Fault Tolerance," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, Mar. 2011, pp. 234–243. doi: 10.1109/ISORCW.2011.42.

[70] P. D. Kaur and K. Priya, "Fault tolerance techniques and architectures in cloud computing - a comparative analysis," in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, Oct. 2015, pp. 1090–1095. doi: 10.1109/ICGCIoT.2015.7380625.

[71] A. Ganesh, M. Sandhya, and S. Shankar, "A study on fault tolerance methods in Cloud Computing," in *2014 IEEE International Advance Computing Conference (IACC)*, Feb. 2014, pp. 844–849. doi: 10.1109/IAdCC.2014.6779432.

[72] A. Ledmi, H. Bendjenna, and S. M. Hemam, "Fault Tolerance in Distributed Systems: A Survey," in *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, Oct. 2018, pp. 1–5. doi: 10.1109/PAIS.2018.8598484.

[73] S. L. Scott *et al.*, "A tunable holistic resiliency approach for high-performance computing systems," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, Feb. 2009, pp. 305–306. doi: 10.1145/1504176.1504227.

[74] T. Tamilvizhi and B. Parvathavarthini, "A novel method for adaptive fault tolerance during load balancing in cloud computing," *Clust. Comput.*, vol. 22, no. 5, pp. 10425–10438, Sep. 2019, doi: 10.1007/s10586-017-1038-6.

[75] M. Hasan and M. S. Goraya, "Fault tolerance in cloud computing environment: A systematic survey," *Comput. Ind.*, vol. 99, pp. 156–172, Aug. 2018, doi: 10.1016/j.compind.2018.03.027.

[76] M. Nazari Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, "A survey of fault tolerance architecture in cloud computing," *J. Netw. Comput. Appl.*, vol. 61, pp. 81–92, Feb. 2016, doi: 10.1016/j.jnca.2015.10.004.

[77] E. AbdElfattah, M. Elkawkagy, and A. El-Sisi, "A reactive fault tolerance approach for cloud computing," in *2017 13th International Computer Engineering Conference (ICENCO)*, Dec. 2017, pp. 190–194. doi: 10.1109/ICENCO.2017.8289786.

[78] L. Guan, H. Chen, and L. Lin, "A Multi-Agent-Based Self-Healing Framework Considering Fault Tolerance and Automatic Restoration for Distribution Networks," *IEEE Access*, vol. 9, pp. 21522–21531, 2021, doi: 10.1109/ACCESS.2021.3055284.

[79] J. Nikolić, N. Jubatyrov, and E. Pournaras, "Self-Healing Dilemmas in Distributed Systems: Fault Correction vs. Fault Tolerance," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 3, pp. 2728–2741, Sep. 2021, doi: 10.1109/TNSM.2021.3092939.

[80] R. Frei, R. McWilliam, B. Derrick, A. Purvis, A. Tiwari, and G. Di Marzo Serugendo, "Self-healing and self-

repairing technologies," *Int. J. Adv. Manuf. Technol.*, vol. 69, no. 5, pp. 1033–1061, Nov. 2013, doi: 10.1007/s00170-013-5070-2.

[81] R. Salvador, A. Otero, J. Mora, E. de la Torre, L. Sekanina, and T. Riesgo, "Fault Tolerance Analysis and Self-Healing Strategy of Autonomous, Evolvable Hardware Systems," in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov. 2011, pp. 164–169. doi: 10.1109/ReConFig.2011.37.

[82] B. Navas, J. Öberg, and I. Sander, "The upset-fault-observer: A concept for self-healing adaptive fault tolerance," in *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Jul. 2014, pp. 89–96. doi: 10.1109/AHS.2014.6880163.

[83] B. Navas, J. Öberg, and I. Sander, "On providing scalable self-healing adaptive fault-tolerance to RTR SoCs," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec. 2014, pp. 1–6. doi: 10.1109/ReConFig.2014.7032541.