Overcoming Byzantine Failures Using Checkpointing

Adnan Agbaria*
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
adnan@crhc.uiuc.edu

Roy Friedman
Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel
roy@cs.technion.ac.il

Abstract

The common approach to masking Byzantine failures is by replicating the computation and running a Byzantine agreement protocol among all replicas. However, Byzantine agreement incurs high communication overhead and also requires the use of at least $3f + 1$ replicas in order to overcome f such failures. However, for many applications, and in particular scientific computation, it is possible to achieve the same goal with much lower average communication and replication overheads. This paper presents a new approach for detecting a Byzantine failure by combining checkpoint/restart with replication. The main benefit of the approach is that when there are no failures, we only use $f + 1$ replicas. If a failure occurs, it is detected using a $3f + 1$-node Byzantine agreement protocol, which also identifies the bad nodes and eliminates them, so the computation can proceed with $f + 1$ replicas until the next failure occurs.

1 Introduction

Byzantine failures are defined as arbitrary deviations of a process from its assumed behavior based on the algorithm it is supposed to be running and the inputs it receives. Such failures can occur, e.g., due to a software bug, a (transitional or permanent) hardware malfunction, or a malicious attack. Overcoming Byzantine failures is becoming increasingly important for several reasons. Increasingly, we depend on the accuracy and correctness of computation in daily life, and the damage caused by a bad computation can have a severe monetary cost. Also, as most computers are connected to the Internet nowadays, they are exposed to hackers, which increases the likelihood of malicious attacks. Finally, recent initiatives, such as GRID computing and peer-to-peer computing, rely on the ability to run computations on foreign computers that cannot be trusted.

Under appropriate synchrony assumptions, it is possible to overcome Byzantine failures by replicating the application on $3f + 1$ nodes, where f is the maximum number of simultaneous failures,

*Supported by DARPA contract no. F30602-00-C-0172.

and running an agreement protocol on each action [8, 12, 19]. Such an approach is required if each action is irrevocable, but is overly expensive in other settings, both in its communication costs and in the number of replicas [26]. Another way to overcome Byzantine failures is by using intrusion detection systems (IDSs) [20, 17]. Upon detection, the faulty process can be removed and may be replaced by another correct process. However, there are many experimental results showing that IDSs can only detect a small fraction of Byzantine failures. In addition, IDSs may produce false alarms that lead to removal of correct processes [23].

In particular, for scientific applications, it is possible to simply run the computation on multiple nodes, and choose the results that repeat more than a threshold number of times. A similar idea is used, e.g., by the SETI@HOME project [1]. However, such an approach is very wasteful, especially if we assume that most computations are failure-free. Moreover, faulty nodes are detected only at the end of the computation, furthers wasting system resources, and the mechanism relies on a single trusted entity to decide which computations are correct.

In this paper, we propose a different approach, which combines checkpoint/restart [3, 14, 24] with replication, and allows a computation to overcome Byzantine failures in deterministic computations assuming operations are revocable. In our approach, each computation is carried by $f + 1$ independent replicas that must also periodically take a checkpoint of their state and store it in a globally accessible location. A different set of $3f + 1$ nodes, called *auditors*, are then used to verify that all $f + 1$ replicas have taken the same checkpoint. Otherwise, the auditors detect the correct checkpoint, eliminate the nodes that reported false checkpoints, and restart the computation from the last correct checkpoint. Clearly, the auditors are shared by many computations; thus, assuming a large pool of nodes and computations, each failure-free computation requires only $f + 1$ nodes. Moreover, even faulty computations require extra nodes only during the phases of the computation in which the failures occurred. Another interesting feature of our approach is that we do not rely on assumptions those of "well-formed message" [21, 18] and "unstolen private keys" [26].

Since our approach is considered a kind of anomaly-based detection, which abnormalities are detected through comparison of the replicas' states, we do not have false alarms, and our mechanism may be better than other IDS systems (such as Snort [2, 23]) at detecting worms.

The remainder of this paper is organized as follows. Section 2 describes the system model and basic definitions. Our approach of anomaly-based detection is presented in Section 3. In Section 4, we describe previous related work, and conclude our work in Section 5.

2 System Model, Assumptions, and Definitions

We consider a distributed system consisting of several processes communicating by sending messages and operating under the *timed asynchronous* model assumptions [10]. In the system model, processes execute a *protocol* and have access to a *local hardware clock* and a *datagram service*. The

local hardware clocks of different processes are not precise, but we assume that their relative drift is bounded. The datagram service is used for communication between the processes. We assume that it only delivers messages that were sent by some particular process, that it delivers a message no more than once, and that it delivers all messages within a timeout δ with high probability (messages are either delivered late or dropped with very low probability). Each process is associated with a *local state*, which includes its local variables and messages that were received from the datagram service. A *computational step* in our model is a function that takes as input the current local state of a process and generates a new local state and a list of messages for the datagram service to send. The protocol specifies the computational steps that should be performed and the messages that must be sent to other processes. We also assume that a process can handle computational steps within a bounded latency with high probability.

Failure to complete a computational step or deliver a message within the expected time bound is considered a *performance failure*. Additional failures that might occur are *crash failures*, in which a process completely fails by not performing any additional computational steps, *omission failures*, in which a message is never delivered, and *Byzantine failures*, in which a process can arbitrarily deviate from its protocol. However, we assume that processes cannot impersonate other processes, i.e., messages are authenticated. Clearly, Byzantine failures include all other forms of failures in the model. A process that suffers a failure is said to be *faulty*; otherwise, it is *correct*. The number of faulty processes is bounded by f.

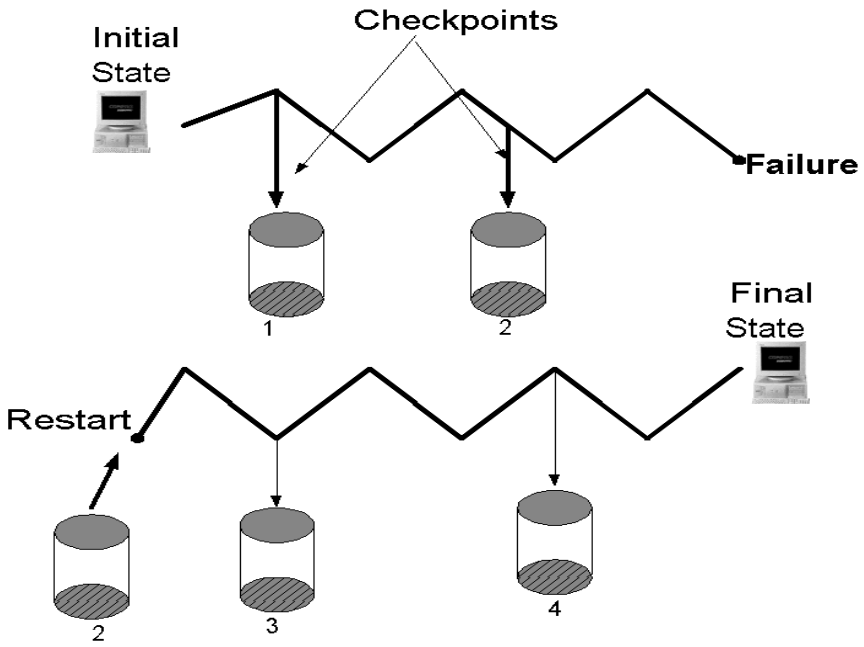

Figure 1: A typical Checkpoint/Restart on an application

As presented in Figure 1, a *checkpoint* is the act of generating a file that includes a process

state during the course of execution of an application. *Restart* is the act of restarting the process from a checkpointed state/file. We assume that the computation tasks are such that a task can be restarted from any checkpoint on any process in the system.

3 Our Approach

In this section, we explain our scheme for overcoming Byzantine failures under the assumptions outlined in Section 2. We designate a fixed set of $3f+1$ processes to be the *auditors*, while all other processes are referred to as *workers*. We define a *computational task* to be a deterministic function and an associated set of input values for that function. In particular, in this work, we assume that a computational task can be completed without exchange of messages with other processes. The goal of the system is to compute the correct results of as many computational tasks as possible. The auditors send computational tasks to workers, which must compute the correct result of applying the function to the corresponding input values. However, completing a computational task may require multiple computational steps.

Due to the possibility of failures, the auditors send the same computation task to $f+1$ workers. We assume that the exact same sequence of computational steps is performed by each worker. In order to monitor the computation, after each predefined number of computational steps, each worker takes a checkpoint of its state, and sends this checkpoint to all auditors. We assume that checkpoints taken at different workers after the same number of computational steps are exactly the same.[1] Given the model assumptions, the auditors know a deadline by which each checkpoint must be delivered to them by each worker. For each checkpoint C_i, the auditors run a Byzantine agreement protocol in which they decide if all workers submitted their checkpoints and if all checkpoints are the same. If the answer is positive, the auditors commit checkpoint C_i.

Otherwise, either the auditors did not receive all checkpoints, or some checkpoints did not agree. If $1 \leq l \leq f+1$ workers failed to send their checkpoints in time to enough auditors, those workers are considered faulty, and the entire computational task, along with the previously committed checkpoint C_{i-1}, is sent to l new workers to continue from there. Otherwise, if some checkpoints did not match, the entire computational task along with C_{i-1} is sent to f additional workers. If that happens, after the new f workers submit their version of C_i, the workers agree on which set of at least $f+1$ workers has submitted the same checkpoints. The corresponding checkpoint is committed as the official C_i. The workers that send different checkpoints are considered faulty, and the auditors never send computational tasks to them again. The computational task is then resumed from C_i on $f+1$ unsuspected workers.

Note that for efficiency, the workers can initially send only a summary (e.g., MD5) of their

[1] If the computation task involves invoking a pseudo-random number generator, we assume that all workers apply the same seed on it.

checkpoint, and then the decided checkpoint can be retrieved from one of the correct workers. The auditors then need to compute the summary of the checkpoint they retrieve from the worker to verify that it is as expected. For simplicity, we present the protocol without this obvious optimization. In addition, if a faulty replica sends its checkpoint to a subset of the auditors, it will not affect the decision made by the auditors. Since the auditors run a Byzantine agreement to reach consensus, the decision remains correct only if there are no more than f faulty auditors.

A serial application is replicated to $f+1$ replicas (processes), denoted by $p_1, \cdots, p_{f+1}$. The set of replicas is denoted by $\mathcal{G}(p)$. We recommend that each process run on a different physical machine, for hardening [26, 29].

In essence, the protocol works as follows. Every T time units, the auditors invoke a checkpoint request to $\mathcal{G}(p)$. Upon a checkpoint request, every replica p_i takes a checkpoint C_{p_i}. Then, every p_i broadcasts $d(C_{p_i})$ to the auditors. After every auditor Λ_i has received all the checkpoint digests from $\mathcal{G}(p)$, the auditors agree on the values of $d(C_{p_i})$ for each replica p_i, $1 \leq i \leq f+1$. If all the checkpoint digests are valid and equivalent, then all the replicas are believed to be correct. Otherwise, the auditors try to find the faulty process(es) for removing them out of $\mathcal{G}(p)$. Actually, the auditors *replay* the last checkpoint interval to determine which processes are faulty.

The protocol starts working upon a checkpoint request to $\mathcal{G}(p)$; during the request, the following steps are performed.

1. Each replica p_i takes a temporary checkpoint C'_{p_i} and then broadcasts a message containing $d(C'_{p_i})$ to Λ.

2. Whenever an auditor Λ_d collects all the messages from all the replicas $\mathcal{G}(p)$ within a time-out, it checks if all the digests are equivalent.

3. The auditors agree on correct digest values. If they agree that the digest values are equivalent, then the checkpoint digests are assumed to be from correct processes.

4. Otherwise, the auditors suspect that there is at least one faulty process in $\mathcal{G}(p)$. Then, they try to detect the faulty process by applying the following steps:

   (a) Each auditor Λ_d restarts a replica of the application from the last committed checkpoint file.

   (b) Each replica Λ_d takes a checkpoint of its replica after T units of time.

   (c) The auditors agree on the correct checkpoint file, say C_Λ.

   (d) Then, they agree on the faulty replicas of $\mathcal{G}(p)$, denoted by F, by comparing C_Λ to C'_{p_i} for all $1 \leq i \leq f+1$.

5. The auditors stop the faulty replicas F.

6. C_{p_i} is committed as the correct checkpoint file.

7. $|F|$ replicas are restored in place of the faulty ones (on different nodes) from C_{p_i}.

Figure 2 presents the pseudo-code of the main function of the protocol. Notice that after the auditors agree to check if there is a faulty process in $\mathcal{G}(p)$, each auditor Λ_d calls the function **verifyLG** with the process p as a parameter.

```
verifyLG (p)
1:  chkptNotify(G(p), 0)
2:  collectMsgs(G(p), 0)
3:  agree(d(C_{p_1}), ..., d(C_{p_{f+1}}))
4:  If all the decided d(C_{p_i}) are equivalent
5:     chkptCommit(C_{p_1})
6:     Return   {There are no suspected faulty processes in G(p) }

    {Λ_d suspects that some processes in G(p) are faulty }
    { Let F be the set of nodes that are suspected to run faulty process(es) }
7:  (F, C) = detectFaulty(G(p))
8:  Stop the faulty replicas F
9:  chkptCommit(C)
10: Restart |F| + 1 replicas from C
```

Figure 2: Auditor Λ_d verifies $\mathcal{G}(p)$.

As depicted in Figure 2, the function **verifyLG** uses several functions. Below we explain each function, and for illustration, we present the pseudo-code of some functions.

- **chkptNotify** - The function **chkptNotify**($\mathcal{G}(p)$, T) tells all the replicas of $\mathcal{G}(p)$ to take a checkpoint immediately after T units of time.
- **collectMsgs** - This function collects the digest values from $\mathcal{G}(p)$ within a time-out.
- **agree** - Upon invoking the function **agree**(v), the auditors apply a consensus protocol [18, 22] to agree on the value of v.
- **chkptCommit -** This function commits the newest checkpoint files of $\mathcal{G}(p)$.
- **detectFaulty** - This function returns the set of the faulty processes and the correct checkpoint as described in Step 4 above.

Notice that after detecting the faulty processes, we can restart the replicas of $\mathcal{G}(p)$ from the last committed checkpoint on different machines to avoid malicious attacks.

3.1 The Protocol Properties

In this section, we briefly discuss the safety and liveness of the protocol. We show that if there are faulty replicas, they will eventually be detected, and then replaced by correct replicas.

Since we have $f+1$ replicas running, there is always at least one correct replica running. If there are some faulty replicas that run out of the normal execution, then after capturing the replicas' state by checkpointing, the auditors will see different states. That makes necessary for us to use $3f+1$ processes (by auditors) to figure out the faulty replicas [8, 27].

In addition, since every checkpoint is committed after a Byzantine agreement has been reached among the $3f+1$ auditors, each checkpoint represents a state of a correct replica. Upon detection of a faulty replica, a recovery is made through use of a committed checkpoint file to restart a new correct replica.

Finally, we would like to point out that most Byzantine agreement protocols, including [18, 22], ensure that if the same value is proposed by all correct processes, then this value will be decided on. This property prevents malicious auditors from forcing the others to execute the **detectFaulty** function when all workers have returned the same checkpoint.

4 Related Work

There is a large body of research on Byzantine fault tolerance for distributed systems [26, 5, 8, 18, 19, 22]. To the best of our knowledge, all of the existing papers assume that there are at least $3f+1$ replicas to tolerate at most f faulty replicas that can occur at the same time, regardless of the number of actual failures. Our work is the first that relies on $f+1$ replicas when there are no failures. However, when a failure occurs, our scheme employs a total of up to $5f+1$ processes ($3f+1$ auditors, $f+1$ workers that execute the task initially, and up to an additional f workers, depending on the number and type of failures, for re-executing the task after a failure).

In [18] Kihlstrom et al. extended the work of Chandra and Toung [9] on unreliable fault detectors for crash faults, by considering unreliable fault detectors for Byzantine faults. They used those detectors to reach consensus in an asynchronous distributed system. Similarly, Malkhi and Reiter [21] solved the consensus problem in asynchronous distributed systems, but unlike the algorithm of Kihlstrom et al., their consensus algorithm relies on a reliable broadcast service. In [8] Castro and Liskov presented a Byzantine fault tolerance algorithm for state machine replication [28]. As in [18, 21] and other papers, their algorithms relay on $3f+1$ replicas. Other papers that solve general Byzantine agreement with unreliable failures detectors that can only detect mute failures include [6, 15].

In [26] Ramasamy et al. extended the Ensemble group communication toolkit [16] to support intrusion tolerance. Originally, Ensemble was designed for crash fault tolerance; Ramasamy et al.

re-implemented the reliable multicast and the group membership protocols to support Byzantine fault tolerance with $3f+1$ replicas. This work was done to provide intrusion tolerance to the ITUA infrastructure [11].

Our work is not the first to combine the techniques of checkpoint/restart and replication for providing intrusion detection. Other projects that use such combinations are Starfish and Manetho. The Starfish system [4] uses such a combination to provide crash fault tolerance for both serial and message-passing applications. The Manetho system [13] gives the application the ability to choose either checkpoint/restart or replication techniques for providing crash fault tolerance.

Several papers [25, 30] used checkpoint/restart and duplication for detecting transient faults. Ziv and Bruck [30] adapted a transient fault-detection mechanism by taking a checkpoint of two replicas and then comparing their states. In addition, Black et al. [7] presented a technique dealing with fail-silent processes in the Voltan application programming environment using self-checking process pairs. Their work was based on hardware-based replication.

5 Conclusions

We have presented a scheme that enables deterministic computations to overcome up to f Byzantine failures using $f+1$ replicas and $3f+1$ auditors. The protocol is based on periodic checkpoints, and assumes that it is permissible to roll back the execution to a previous checkpoint. Our work so far does not address collaborative applications among several computing processes that are not replicas of the same program, as we assume that all replicas receive the same set of inputs. In the future, we intend to extend our protocol in that direction (i.e., to support distributed or parallel computations) and measure its costs empirically.

Acknowledgments

We would like to thank William H. Sanders for his helpful comments and Jenny Applequist for her editorial assistance.

References

[1] SETI@home Home Page. http://setiathome.ssl.berkeley.edu.

[2] The Snort Home Page. http://www.snort.org.

[3] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg. Quantifying Rollback Propagation in Distributed Checkpointing. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*, pages 36–45, New Orleans, USA, October 2001.

[4] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, August 1999.

[5] L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter. Fault Detection for Byzantine Quorum Systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):996–1007, September 2001.

[6] R. Baldoni and J. M. Helary. Strengthening Distributed Protocols to Handle Tougher Failures. Technical Report #1477, IRISA, Université de Rennes, France, 2002.

[7] D. Black, C. Low, and S. K. Shrivastava. The Voltan Application Programming Environment for Fail-silent Processes. In *Distributed Systems Engineering*, pages 66–77, June 1998.

[8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, February 1999.

[9] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[10] F. Cristian and C. Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[11] M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett. Intrusion Tolerance Approaches in ITUA. In *Fast Abstract in Supplement of the 2001 International Conference on Dependable Systems and Networks*, pages B–64–B–65, Goteborg, Sweden, July 2001.

[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, April 1988.

[13] E. N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. PhD thesis, Department of Computer Science, Rice University, 1993.

[14] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Department of Computer Science, Carnegie Mellon University, June 1999.

[15] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and Efficient Oracle-Based Consensus Protocols for Asynchronous Systems. Technical Report 1556, IRISA - Institute de Recherche en Informatique et Systemes Aleatoires, September 2003.

[16] M. Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, January 1998. Also published as Cornell Dept. of CS Technical Report no. TR98-1662.

[17] H. S. Javitz and A. Valdes. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Conference on Research in Security and Privacy*, pages 316–376, Oakland, CA, May 1991.

[18] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[20] Ulf Lindqvist. *On the Fundamentals of Analysis and Detection of Computer Misuse*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1999.

[21] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW97)*, pages 116–124, Rockport, MA, June 1997.

[22] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *The Journal of Distributed Computing*, 11(4):203–213, 1998.

[23] D. Newman, J. Snyder, and R. Thayer. Crying Wolf: False Alarms Hide Attacks. In *Network World*. Network World, Inc., June 2002.

[24] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.

[25] D. K. Pradhan. Redundancy Schemes for Recovery. Technical Report TR-89-cse-16, ECE Department, University of Massachusetts, Amherst, 1989.

[26] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 229–238, Washington, DC, June 2002.

[27] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[28] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[29] T. Wu, M. Malkin, and D. Boneh. Building Intrusion-Tolerant Applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, 1999.

[30] A. Ziv and J. Bruck. Efficient Checkpointing Over Local Area Networks. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 30–35, June 1994.