Efficient failure detection and consensus at extreme-scale systems

Soma Sekhar Kolisetty, Battula Srinivasa Rao
School of Computer Science and Engineering, VIT-AP University, Amaravati, Andhra Pradesh, India

Article Info

Article history:

Received Nov 5, 2021
Revised May 20, 2022
Accepted Jun 14, 2022

Keywords:

Consensus
Epidemic protocol
Failure detection
Fault tolerance
Gossip-based protocol
Message passing interface process
Scalability

ABSTRACT

Distributed systems and extreme-scale systems are ubiquitous in recent years and have seen throughout academia organizations, business, home, and government sectors. Peer-to-peer (P2P) technology is a typical distributed system model that is gaining popularity for delivering computing resources and services. Distributed systems try to increase its availability in the event of frequent component failures and functioning the system in such scenario is notoriously difficult. In order to identify component failures in the system and achieve global agreement (consensus) among failed components, this paper implemented an efficient failure detection and consensus algorithm based on fail-stop type process failures. The proposed algorithm is fault-tolerant to process failures occurring before and during the execution of the algorithm. The proposed algorithm works with the epidemic gossip protocol, which is a randomly generated paradigm of computation and communication that is both fault-tolerant and scalable. A simulation of an extreme-scale information dissemination process shows that global agreement can be achieved. A P2P simulator, PeerSim, is used in the paper to implement and test the proposed algorithm. The proposed algorithm results exhibited high scalability and at the same time detected all the process failures. The status of all the processes is maintained in a Boolean matrix.

This is an open access article under the CC BY-SA license.

Corresponding Author:

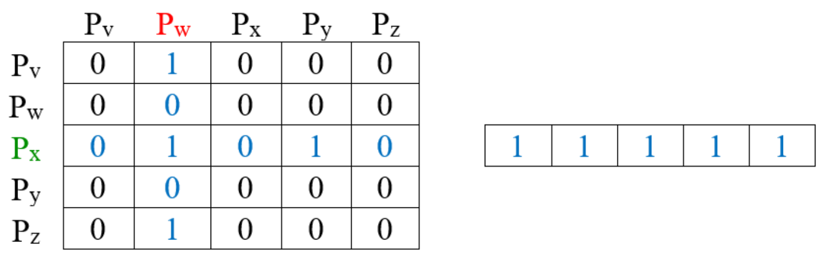
Battula Srinivasa Rao
School of Computer Science and Engineering, VIT-AP University
Amaravati-522 237, Andhra Pradesh, India
Email: sreenivas.battula@gmail.com

1. INTRODUCTION

A collaborative fault-tolerance approach is essential to diminish frequent component failures efficiently to ensure extreme-scale systems success. Many fields of distributed systems have adopted algorithm-based fault tolerant procedures to detect failures in the system [1]. In research [2], [3] the authors have proposed a replication model and load balancing algorithm that provide fault tolerance in large-scale systems using gossiping with node failures. A gossip-based parallel algorithms are introduced for high-performance computing, unstructured peer-to-peer networks and decentralized job scheduling in grids [4]–[6]. Fault tolerant techniques for message passing interface (MPI) applications have been proposed to detect and recover failures [7]–[11]. Failure detection using machine learning frameworks in high performance computing systems that automatically detect and diagnose failures [12]–[14]. A state-of-the-art failure detection, prediction, and recovery techniques in exascale systems has introduced [15]. In research [16], [17], a fault tolerant framework is designed and implemented for heterogeneous applications to increase scalability. A various gossip-based algorithms were implemented for failure detection and consensus [18]–[20].

This paper particularly focuses on [1] in improving the scalability of the first algorithm with the help of an efficient failure detection and consensus algorithm. The algorithm presented in this work is completely fault tolerant. It detects process failures and consensus accurately using gossip protocol. For the purpose of detecting failures in large-scale systems, gossiping has been used to maintain up-to-date information about process status. P2P systems can be large scale (millions of nodes) extremely. Process in such systems is continuously joining and leaving. In this frame, experimenting with a protocol can jeopardizes the efficiency of the system. In order to cope with these properties, PeerSim simulator was used, in enabling maximum scalability and dynamism.

Consensus is detected with the help of alive processes using the same approach of gossiping by preserving the status of all processes in a matrix. Every process in the system maintains the status of other processes in a fault matrix M_q-holds i^2 elements where i is the number of processes in the system. 0 indicate process is alive. 1 indicate process have failed. Five processes P_v-P_z are shown in Figure 1. We can check for consensus at any one of the alive processes say P_x. We can detect a particular process say P_w is alive or failed with other alive processes P_v, P_w, P_y, and P_z by overlapping process P_x with process P_w which is shown in Figure 1 separately. Hence, in this case, consensus is detected by process P_x for process P_w. The size of the fault matrix increases exponentially with the system size.



	P_v	P_w	P_x	P_y	P_z
P_v	0	1	0	0	0
P_w	0	0	0	0	0
P_x	0	1	0	1	0
P_y	0	0	0	0	0
P_z	0	1	0	0	0

1	1	1	1	1

Figure 1. Process P_w's consensus detection

Bosilca *et al.* [21] presented a scalable method that can tolerate failures of high-frequency and demonstrated a hypothetical upper bound just on time needed to alter the system that permits novel failures to occur in a state. This means that the algorithm is tolerant of a random failures number, so long as they do not appear more frequently. This method was evaluated on large-scale applications using a resilient MPI distribution. For reasonable detection times in the system, the ring strategy introduces non-measurable or insignificant extra noise, whereas the strategy of reliable broadcast for alerts permits for efficient delivery of flaw information once discovered.

Emani *et al.* [22] suggested a new approach: checkpointable MPI, in which the MPI state is recorded in a checkpoint with the state of application. MPI and application states are usually recovered from the most recent checkpoint during the event of a failure. Among the benefits of such a framework are the limited recovery time, the retention of the MPI state prior to a given failure, the similarity to existing application checkpoint and restart mechanisms, and the ease of integrating with large-scale scientific applications.

Losada *et al.* [23] developed a solution for local rollback of MPI applications that integrates a number of resilience techniques. With user level failure mitigation (ULFM), failure mitigation builds coupled with a tool of compiler-driven application-level checkpointing (CPPC), as well as the logging of message facilities of OpenMPI, vProtocol pessimistic message logging, significantly less resilience overhead was generated. As a result of the ULFM resilience attributes, breakdowns have been identified, communications infrastructure managed among survivor processes, replaced processes have re-spawned, and communicators have been rebuilt. The last checkpoint enabled the recovery of failed processes, while message logging allowed further computations to be progressed.

Using the new ULFM features, Losada *et al.* [24] extended the CPPC checkpointing framework. A transparent method for acquiring flexible MPI applications was provided by implementing the original code of application. Further, multithreaded multiple level check pointing improved scalability by storing checkpoints in separate memory levels. Tolerating failures in MPI processes resulted in low overhead.

Shahzad *et al.* [25] presented automatic fault tolerance and checkpoint-restart (CRAFT), a C++ library that provided two functions. The library can easily be expanded to include more data types. CRAFT simplifies the interface for ULFM-based recovery of dynamic process, reducing the effort and complexity required to detect faults as well as recover from faults. A combination of application-level checkpoints and dynamic process recovery allows applications to write application-level checkpoints with small effort. This project detailed the design and use of the library. For an in-depth analysis of the associated overheads, several benchmarks were employed.

In our proposed approach, we developed an efficient-scalable failure detection and consensus algorithm that has implemented for detecting failures and achieving consensus among the failed processes using a gossip-style protocol. Rest of the sections are: Research method is exhibited in section 2. Whereas section 3 deals with performance analysis and section 4 concludes the paper.

2. RESEARCH METHOD

The proposed algorithm uses a gossip-based protocol is discussed in this section as a method for detecting MPI process failures. During the execution, every process periodically pings another random process to detect failures. In addition, achieving consensus on the list of failed processes while maintaining the global knowledge at each process for the dissemination of information.

The proposed algorithm has the following assumptions: a reliable mode of communication is assumed to connect processes, there is an assumption that a failed MPI process ceases communication, a failed MPI process cannot be recovered since faults are assumed permanent, if a process is detected as failed, all the other processes will detect that process has failed, the consensus algorithm will stop at some point for a prolonged period to attain consensus.

2.1. Failure detection approach

In this section, we proposed how failure detection takes place using randomized pinging. Randomized pinging is closely related to gossip-based protocols. They are especially useful and efficient in distributed systems where processes are only aware of their local connections and do not have global knowledge. By executing these gossip algorithms, the processes can work together to achieve the goal of collecting and disseminating the information across the entire system.

Pseudo-code for the algorithm can be showed in Figure 2. The process *u* pings a process *v* at random, based on a uniform probability distribution function, during the length of a T_{Gossip} cycle. As long as process *v* responds to process *u* before the termination of the current gossip cycle, then process *u* discovers process *v* as alive; if not, process *v* is considered as fail.

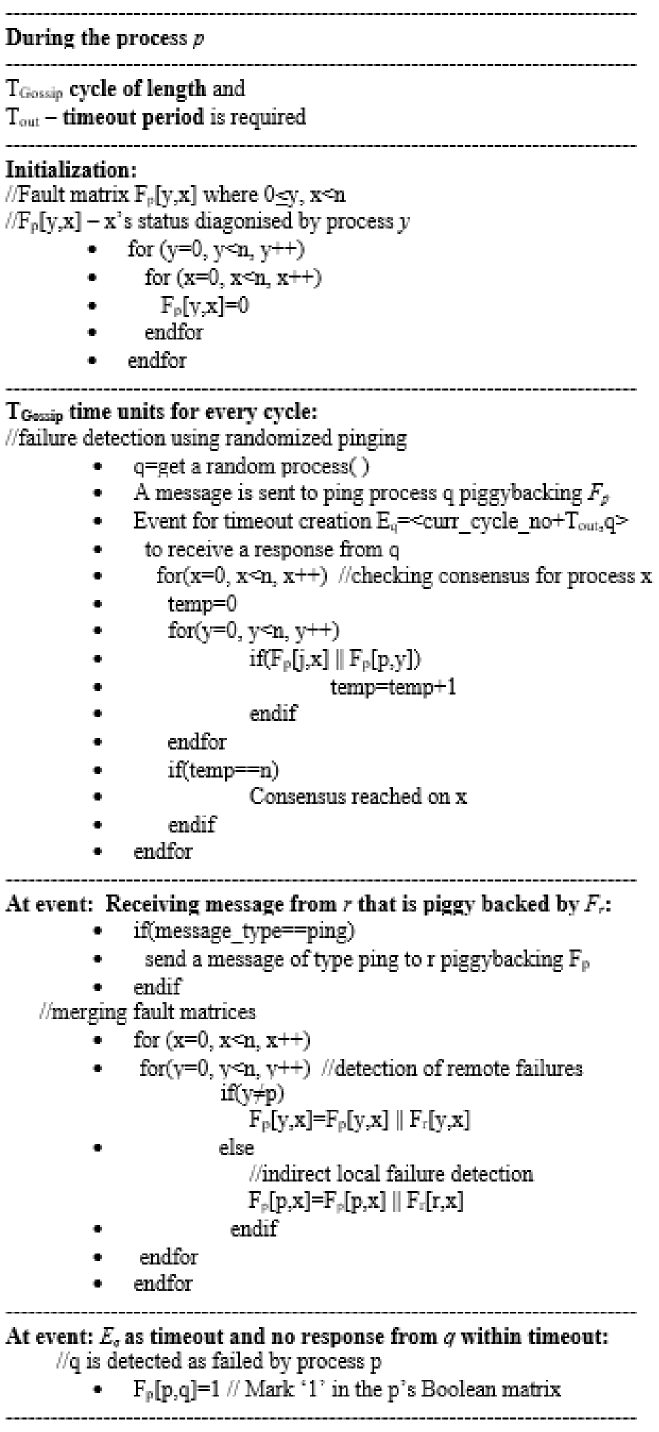
The probability of the node being chosen as the destination of zero, a ping message or several ping messages continues to follow a binomial distribution during a uniform gossiping cycle, thus making failure detection more scalable and putting a minimum burden on each process. Failures are inevitably detected by the non-failed processes, causing an exponential transmission of communication to reach concurrence. A gossip-based technique is fault-tolerant, so it can handle delays and moderate to low message loss rates.

```
---------------------------------------------------------------------------------
At each process u
---------------------------------------------------------------------------------
At every cycle (each T_Gossip time units):
        • v=getRandomProcess( )
        • ping v with a message
---------------------------------------------------------------------------------
At event: ping message was received from v:
        • Respond to v by sending response message
---------------------------------------------------------------------------------
At event:  without getting a message from v tends to timeout:
        • mark v as failure
---------------------------------------------------------------------------------
```

Figure 2. Efficient failure detection

2.2. Global knowledge for consensus

We can achieve consensus using consensus algorithm by retaining global knowledge within every MPI process. Upon detecting failures, a fault matrix F_p is maintained by each process *p* to assist in the memory of all process's status, as believed by it, as well as by all other processes, as depicted in Figure 3. Process *u* detects a failure of process *v* in its fault matrix and is represented as: F_p[u,v]. (1 if detected to have failed; 0 otherwise). Four logical tasks can be distinguished depending on the tasks performed: i) initializing, ii) detecting process failures, iii) updating fault matrix, and iv) checking for consensus.



```
---------------------------------------------------------------------------------
During the process p
---------------------------------------------------------------------------------
T_Gossip cycle of length and
T_out – timeout period is required
---------------------------------------------------------------------------------
Initialization:
//Fault matrix F_p[y,x] where 0≤y, x<n
//F_p[y,x] – x's status diagonised by process y
    •   for (y=0, y<n, y++)
    •     for (x=0, x<n, x++)
    •       F_p[y,x]=0
    •     endfor
    •   endfor
---------------------------------------------------------------------------------
T_Gossip time units for every cycle:
//failure detection using randomized pinging
    •   q=get a random process( )
    •   A message is sent to ping process q piggybacking F_p
    •   Event for timeout creation E_q=<curr_cycle_no+T_out,q>
    •     to receive a response from q
    •       for(x=0, x<n, x++)  //checking consensus for process x
    •         temp=0
    •         for(y=0, y<n, y++)
    •               if(F_p[j,x] || F_p[p,y])
    •                       temp=temp+1
    •               endif
    •         endfor
    •         if(temp==n)
    •               Consensus reached on x
    •         endif
    •   endfor
---------------------------------------------------------------------------------
At event:  Receiving message from r that is piggy backed by F_r:
    •   if(message_type==ping)
    •     send a message of type ping to r piggybacking F_p
    •   endif
  //merging fault matrices
    •   for (x=0, x<n, x++)
    •    for(y=0, y<n, y++)  //detection of remote failures
                  if(y≠p)
                     F_p[y,x]=F_p[y,x] || F_r[y,x]
    •             else
                     //indirect local failure detection
                     F_p[p,x]=F_p[p,x] || F_r[r,x]
    •              endif
    •    endfor
    •   endfor
---------------------------------------------------------------------------------
At event: E_q as timeout and no response from q within timeout:
    //q is detected as failed by process p
    •   F_p[p,q]=1 // Mark '1' in the p's Boolean matrix
---------------------------------------------------------------------------------
```

Figure 3. Efficient consensus detection

Each process assumes that all other processes have already started and that none have detected before the algorithm starts. Each T_{Gossip} time unit, a process *p* picks a process q at random and disseminate pings message to it, piggybacking F_p which is the fault matrix of process *p*. It allows information to spread exponentially not only by propagating the entire matrix, but also by propagating the detections of other processes that process *p* is aware of, which allows more efficient communication. When this ping message is received, an asynchronous response is sent. If *p* has not received a response message from *q* at the time of current gossip cycle comes to an end, *q* is directly detected as failed. An event timeout is being set in the current gossip cycle so that *q* can provide a response.

In the local fault matrix F_p, the OR function is applied to the relating aspects in F_r, except for the row p, when a response message is received at position p from position r. The consensus is reached when all the alive processes have detected a failed process. The algorithm requires $O(n^2)$ memory in keeping the local state knowledge by all the processes.

3. PERFORMANCE EVALUATION

Using PeerSim simulator, we have implemented and tested the failure detection and consensus algorithm through simulations. PeerSim is a Java-based P2P network simulator. In order to simulate and test a P2P network in real-life, the simulator excels in creating maximum number of processes (i.e., up to 10 lakh). For EU projects, the DELIS, and BISON, the PeerSim simulator was launched. The PeerSim has been improved with intense dynamism and scalability. It comprises of two simulation engines, the first one is the event-driven engine and the second one is the cycle-driven engine.

In this paper, the major purpose for employing the PeerSim simulator rather than diverse simulators (e.g., NS3) is due to, it supports dynamism, maximal flexibility and enables scalability. The simulator can cope up with millions of nodes and allow to experiment with a protocol and thus enabling extreme scalability. Thus, the failure detection and consensus algorithms were implementation using PeerSim simulator. In the following sections, we define the infrastructure first, and then presented the evaluation results.

3.1. Simulation infrastructure

PeerSim is a Java-based simulator, and it can be used to generate and observe process failures. Also, it can be extended for more sophisticated scenarios and functionalities. In this work, a traditional event-based simulation engine components of PeerSim are used to simulate failure injection and observation. Mainly, the following Java classes were extended: i) node class added more fields for data recording and collection, ii) network class added more accessor methods for the modified node class, iii) dynamic network class added time-slots phenomena to generate failure events at random times during simulation cycles, and iv) observer class is used to collect data, write it into text files, and compute statistical results for plotting using Gnuplot tool. The experiment parameters for protocol, control, and PeerSim objects can be set up in the configuration file. The simulations are run with a configuration file.

3.2. Results and discussion

3.2.1. Algorithm analysis

The fault tolerance and scalability of the proposed consensus algorithm and failure detection were evaluated. A peer-to-peer (P2P) simulator and PeerSim is used to implement and test the proposed algorithm. The processes' fault matrix generated by the proposed algorithm increases with its system size and sending it as part of gossiping utilizes a high amount of memory.

The proposed algorithm detects failures using ping-response information exchange throughout each gossip cycle. A gossip cycle has a binomial distribution for each ping message received by the process. The possibility of a process receiving multiple pinging messages is extremely down, as is the possibility of a failed process not receiving a ping message. Thus, any process failure is detected within the first few gossip cycles.

Ping-reply gossip messages are sent to the system's fault-free processes when a failure is detected. It is observed that the consensus has reached after a logarithmic number of gossip cycles. As a result of the ping and response messages in a gossip cycle, messages are disseminated twice as fast. Since the algorithm require two gossip messages (ping-reply) per cycle to detect consensus at each process, the number of gossip messages required for detection is twice what is actually required. It takes the proposed algorithm n^2 memory units per process to store the fault matrices, as is each process in the system will store the fault matrix.

3.3. Experimental results

Experiments are conducted using the PeerSim simulator configured in the Eclipse Luna version. PeerSim is adaptable, scalable, and simple to set up. Gossip-based protocols distribute information exponentially, so enough processes can reach a consensus on failed injections within a short time. Failures were injected into randomly selected simulated MPI processes. A failure is detected/committed at different points in the process cycle (s). Thus, the last process that accomplished this is recorded. The aforementioned facts have all been verified through experiments.

3.3.1. Global knowledge consensus

We set the gossip cycle length that allowing the matrices to merge completely within a given cycle for a given system size. The cycle length was 10; 100; 1,000; 10,000; 100,000 ms respectively for system sizes {8, 16, 32, 64, 128}, {256, 512}, {1024, 2048, 4096}, {8192, 16384, 32768, 65536} and {131072, 262144, 524288, 1000000}. This is required because matrix merge operations take up a significant portion of the cycle time. We conducted experiments to test the algorithm's fault tolerance and scalability.

Failures are injected before the failure detection and makes to run consensus algorithm. Figure 4 displays the relationship between the number of cycles required to achieve consensus and the size of the system for one failure injected before the algorithm. The number of cycles required to reach consensus is obviously proportional to the system size. Figure 5 depicts how failure detection information exponentially spreads for a specific injected failure. The algorithm's logarithmic complexity is demonstrated in both figures.

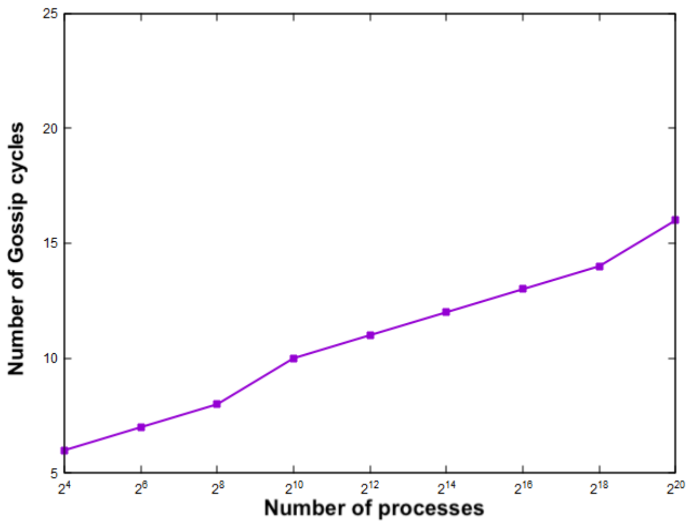


Figure 4. Achieving global agreement after injecting a single failure

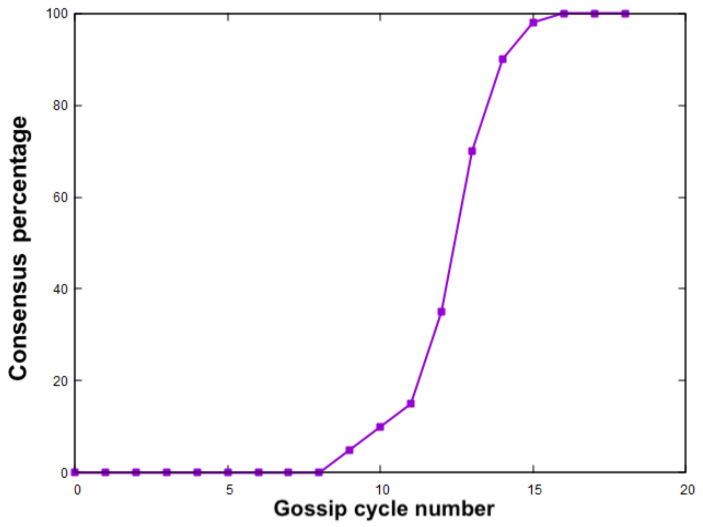


Figure 5. For size of the system 2^{20} progression to local consensus after a single failure injection

Figure 6 shows how different processes reach consensus at different cycle numbers. Figure 7 shows the effect of multiple (eight) failures injected before the algorithm on the consensus time. Compared to the single failure case, it took only one or two more cycles. Failures were injected into the algorithm during execution to test its fault tolerance. In Figure 8, multiple failures were injected at random into randomly selected processes. The number of cycles required to reach consensus has increased gradually. Despite its fault tolerance, the algorithm is completely reliable.

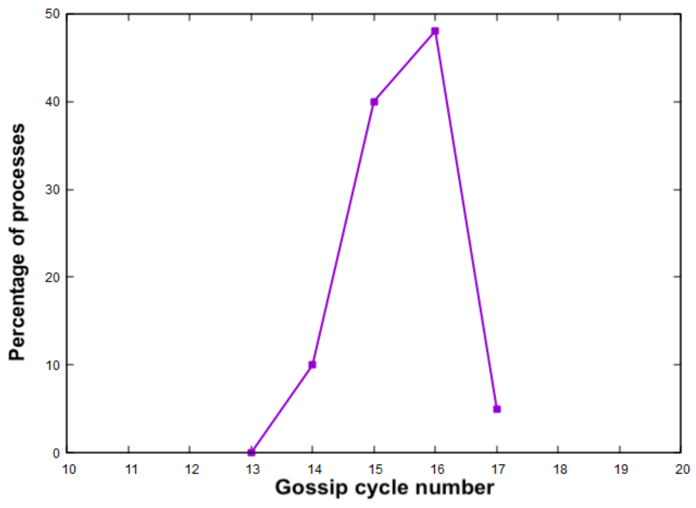


Figure 6. Spreading consensus detection for 2^{20} system size

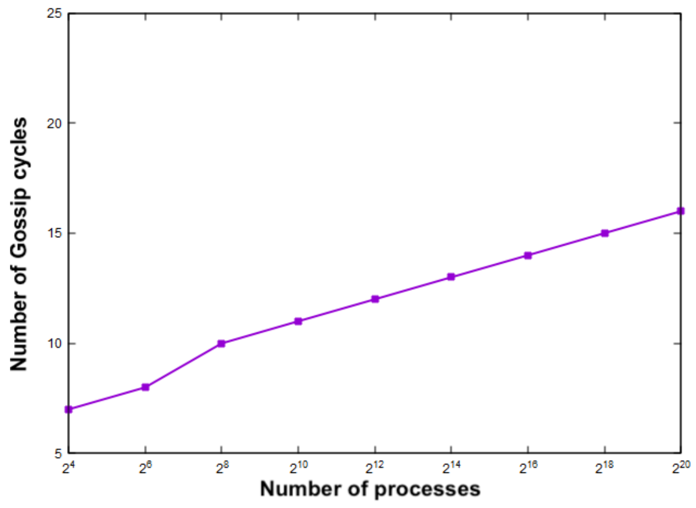


Figure 7. Gossip cycles to obtain global consensus before multiple failures injection

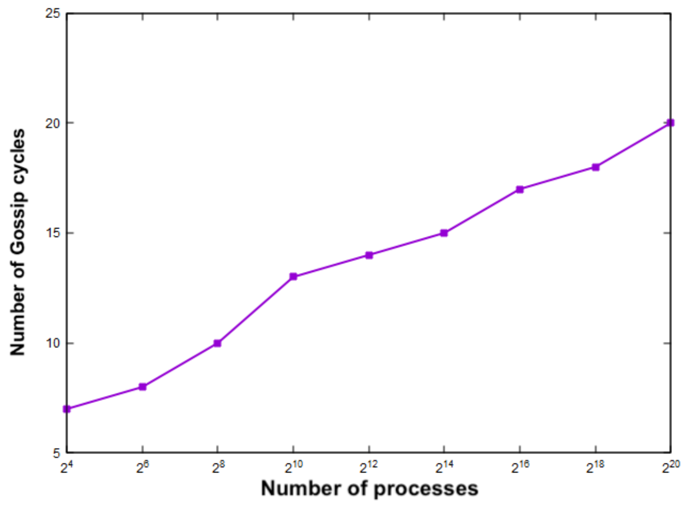


Figure 8. Injected multiple failures during execution of algorithm to achieve global consensus

4. CONCLUSION

An efficient failure detection and consensus algorithm is presented to increase the scalability. As system size increases at scale, experimental results show logarithmic scalability and nearly constant fault detection and consensus time. Fault tolerance is implemented by the algorithm, which succeeds even if process failures are experienced during execution and detects both prior and subsequent failures. The proposed algorithm is efficient as it uses least amount of time for failure detection and achieving consensus. To improve scalability of the algorithm, as is implemented with a Boolean matrix at every process in the system. Previously, simulations were typically conducted with hundreds of nodes. The simulation has scaled up to ten lakh nodes by incorporating resource discovery mechanisms into PeerSim simulator.

REFERENCES

[1] A. Katti, G. Di Fatta, T. Naughton, and C. Engelmann, "Epidemic failure detection and consensus for extreme parallelism," *The International Journal of High Performance Computing Applications*, vol. 32, no. 5, pp. 729–743, Sep. 2018, doi: 10.1177/1094342017690910.

[2] M. Chatterjee, A. Mitra, S. Roy, and S. K. Setua, "Gossip based fault tolerant protocol in distributed transactional memory using quorum based replication system," *Cluster Computing*, vol. 23, no. 2, pp. 1103–1124, Jun. 2020, doi: 10.1007/s10586-019-02973-7.

[3] M. Chatterjee, A. Mitra, S. K. Setua, and S. Roy, "Gossip-based fault-tolerant load balancing algorithm with low communication overhead," *Computers and Electrical Engineering*, vol. 81, p. 106517, Jan. 2020, doi: 10.1016/j.compeleceng.2019.106517.

[4] M. Casas, W. N. Gansterer, and E. Wimmer, "Resilient gossip-inspired all-reduce algorithms for high-performance computing: Potential, limitations, and open questions," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 366–383, Mar. 2019, doi: 10.1177/1094342018762531.

[5] R. Azimi and H. Sajedi, "Peer sampling gossip-based distributed clustering algorithm for unstructured P2P networks," *Neural Computing and Applications*, vol. 29, no. 2, pp. 593–612, Jan. 2018, doi: 10.1007/s00521-017-3119-0.

[6] J. Wu and X. Xu, "Decentralised grid scheduling approach based on multi-agent reinforcement learning and gossip mechanism," *CAAI Transactions on Intelligence Technology*, vol. 3, no. 1, pp. 8–17, Mar. 2018, doi: 10.1049/trit.2018.0001.

[7] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, "Fault tolerance of MPI applications in exascale systems: The ULFM solution," *Future Generation Computer Systems*, vol. 106, pp. 467–481, May 2020, doi: 10.1016/j.future.2020.01.026.

[8] N. Sultana, M. Rüfenacht, A. Skjellum, I. Laguna, and K. Mohror, "Failure recovery for bulk synchronous applications with MPI stages," *Parallel Computing*, vol. 84, pp. 1–14, May 2019, doi: 10.1016/j.parco.2019.02.007.

[9] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit++: evaluating the performance of global-restart recovery methods for MPI fault tolerance," *arxiv.org/abs/2102.06896*, Feb. 2021.

[10] N. Losada, M. J. Martín, and P. González, "Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 316–329, Jan. 2017, doi: 10.1007/s11227-016-1863-z.

[11] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *The International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, Aug. 2004, doi: 10.1177/1094342004046045.

[12] O. Tuncer *et al.*, "Online diagnosis of performance variation in HPC systems using machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 883–896, Apr. 2019, doi: 10.1109/TPDS.2018.2870403.

[13] B. Mohammed, I. Awan, H. Ugail, and M. Younas, "Failure prediction using machine learning in a virtualised HPC system and application," *Cluster Computing*, vol. 22, no. 2, pp. 471–485, Jun. 2019, doi: 10.1007/s10586-019-02917-1.

[14] G. Zhang, Y. Liu, H. Yang, and D. Qian, "Efficient detection of silent data corruption in HPC applications with synchronization-free message verification," *The Journal of Supercomputing*, vol. 78, no. 1, pp. 1381–1408, Jan. 2022, doi: 10.1007/s11227-021-03892-4.

[15] R. Canal *et al.*, "Predictive reliability and fault management in exascale systems," *ACM Computing Surveys*, vol. 53, no. 5, pp. 1–32, Oct. 2020, doi: 10.1145/3403956.

[16] C. Chen, Y. Du, K. Zuo, J. Fang, and C. Yang, "Toward fault-tolerant hybrid programming over large-scale heterogeneous clusters via checkpointing/restart optimization," *The Journal of Supercomputing*, vol. 75, no. 8, pp. 4226–4247, Aug. 2019, doi: 10.1007/s11227-017-2116-5.

[17] A. M. Aseeri and A. Mai, "A two-level fault-tolerance technique for high performance computing applications," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 12, 2018, doi: 10.14569/IJACSA.2018.091207.

[18] T. Yu and J. Xiong, "Distributed consensus-based estimation and control of large-scale systems under gossip communication protocol," *Journal of the Franklin Institute*, vol. 357, no. 14, pp. 10010–10026, Sep. 2020, doi: 10.1016/j.jfranklin.2020.07.019.

[19] G. Wang, Z. Wang, and J. Wu, "A local average broadcast gossip algorithm for fast global consensus over graphs," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 301–309, Nov. 2017, doi: 10.1016/j.jpdc.2017.05.008.

[20] D. Silvestre, P. Rosa, J. P. Hespanha, and C. Silvestre, "Stochastic and deterministic fault detection for randomized gossip algorithms," *Automatica*, vol. 78, pp. 46–60, Apr. 2017, doi: 10.1016/j.automatica.2016.12.011.

[21] G. Bosilca *et al.*, "A failure detector for HPC platforms," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 139–158, Jan. 2018, doi: 10.1177/1094342017711505.

[22] M. Emani, I. Laguna, K. Mohror, N. Sultana, and A. Skjellum, "Checkpointable mpi: A transparent fault-tolerance approach for MPI (No. LLNL-CONF-739586)." Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), 2017.

[23] N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín, "Local rollback for resilient MPI applications with application-level checkpointing and message logging," *Future Generation Computer Systems*, vol. 91, pp. 450–464, Feb. 2019, doi: 10.1016/j.future.2018.09.041.

[24] N. Losada, I. Cores, M. J. Martín, and P. González, "Resilient MPI applications using an application-level checkpointing framework and ULFM," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 100–113, 2017, doi: 10.1007/s11227-016-1629-7.

[25] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 501–514, Mar. 2019, doi: 10.1109/TPDS.2018.2866794.

BIOGRAPHIES OF AUTHORS

Soma Sekhar Kolisetty received his M.Tech. in Software Engineering and B.Tech. in Computer Science and Engineering from Jawaharlal Nehru Technological University, Hyderabad, in 2011 and 2009. Currently, he is pursuing his Ph.D. at VIT-AP University, Amaravati, Near Vijayawada, Andhra Pradesh. He had an academic, industrial and research experience of 8 years. His research interests include distributed systems, parallel computing, machine learning, and data science. He can be contacted at email: sekhar.soma007@gmail.com.

Battula Srinivasa Rao working as Associate professor in School of Computer Science and Engineering, Vellore Institute of Technology-Andhra Pradesh (VIT-AP) University, Amaravati, Near Vijayawada, Andhra Pradesh. His research interests are soft computing, image processing, machine learning and deep learning. He can be contacted at email: sreenivas.battula@gmail.com.