

# Scalable epidemic message passing interface fault tolerance

Soma Sekhar Kolisetty, Battula Srinivasa Rao

School of Computer Science and Engineering, VIT-AP University, Near Vijayawada, Andhra Pradesh, India

## Article Info

### Article history:

Received Nov 15, 2021

Revised Feb 10, 2022

Accepted Feb 23, 2022

### Keywords:

Consensus

Epidemic protocols

Failure detection

Fault tolerance

MPI

Parallelism

Scalability

## ABSTRACT

Resilience and fault tolerance are challenging tasks in the field of high performance computing (HPC) and extreme scale systems. Components fail more often in such systems, results in application abort. Adopting fault-tolerance techniques can be consistently detect failures and continue application's execution even if the failures exist. A prominent parallel programming specification, message passing interface (MPI), as it would be used to implement failure detection and consensus algorithm in this paper. Although the MPI does not facilitate fault tolerant behavior, this work presents a fault tolerant, matrix based failure detection and consensus algorithm. The proposed algorithm uses Gossiping. To detect failures, randomised ping-pong will be applied during the execution of the algorithm by using piggybacked gossip messages. In order to achieve consensus on the failures in the system, failed processes' information will be sent using the same piggybacked gossip messages to all the alive processes. The algorithm was implemented in MPI framework and is completely fault tolerant. The results exhibit all the MPI process failures were detected using randomised ping-pong and global consensus has achieved on failed MPI process in the system.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



## Corresponding Author:

Battula Srinivasa Rao

School of Computer Science and Engineering, VIT-AP University

Near Vijayawada-522 237, Andhra Pradesh, India

Email: [sreenivas.battula@gmail.com](mailto:sreenivas.battula@gmail.com)

## 1. INTRODUCTION

Large-scale and extreme-scale systems are required to work under component failures very often. Fault tolerance approach is essential for future extreme, large system sizes which fail more frequently due to components (node and process) failure. Such systems continue to increase components count, individual component reliability decreases and software complexity increase [1]. To ensure parallel application correctness and execution efficiency in the realm of large scale distributed systems, frequent failures must overcome. Complete application is aborted due to frequent component failures. Adopting fault tolerant, scalable failure detection and consensus approaches will allow to continue the application's execution even in the presence of failures.

Epidemic failure detection [2] is one of the foundation of fault tolerance in distributed systems. Failure detection can take place through gossiping known as *gossip-based failure detection* in which each process announces its aliveness to its neighbour processes frequently. With this notion, every process in the system will come to know about every other process and decide whether a process is alive or failed. This information is gradually disseminated through the network using the same *Gossiping*. In the field of fault-tolerant computing, the consensus problem is the formation of an agreement may be made on any value among the fault-free processes in order to keep up the integrity and performance of the system [3]. The

notion of consensus (agreement) is to share information within a group of processes, ideally in a fault tolerant way i.e. the fault free processes should be able to agree on and give accurate results.

The details of the consensus problem is illustrated in [3]-[5] (chapter 14, section 14.1.1). The paper [6] contributes two state-of-the-art Gossip based algorithms that use randomized ping to detect all the process failures before and during the execution of the algorithms at high speed during Gossip cycles. Failures are circulated to each of the alive processes and consensus is being achieved on the failed processes with the help of consensus algorithms. Implementation and testing are performed with an extreme scale simulator. Consensus is being detected correctly by all the alive processes when they identify the existence of failed processes in the system.

A part from failure detection and achieving consensus, disseminating global information and computing is also a challenging task in large scale, distributed systems [7]. Fault tolerant approaches are appropriate for this task to avoid bottlenecks and failures. Several aggregation protocols were developed and are broadly classified as: (i) tree based protocols and (ii) epidemic protocols. Due to the use of randomized communication model, epidemic protocols are robust, scalable and support maximum number of communications contrast to tree based protocols. Epidemic protocols have the advantage of spreading the information at high speed without additional communication overhead which are inherently fault tolerant. An effective failure detection in large, distributed systems can be done by means of Gossip based protocols. In [8], a Gossip based failure detection and consensus algorithm is examined to mitigate resource utilization and consensus time. Due to its resilient nature, *random gossiping* among the nodes has been investigated for detecting failures in the system. Moreover, the Gossip protocols have the capability to scale up the processes count.

Ayiad and Fatta [9] proposed an epidemic consensus protocol to achieve both global agreement (consensus) among all the nodes from local computation using a decentralised data aggregation. It is composed of four phases: *Aggregation* phase, *Convergence* phase, *Agreement*, and *Commit* phase. To function epidemic consensus protocol, two more protocols: node cache protocol and system size estimation protocol are used to achieve global agreement. The proposed work uses a single epidemic protocol to achieve consensus on MPI process failures. Katti and Lilja [10] proposed a combined epidemic failure detection and consensus algorithm which is acceptable for a very large scale systems. The *PING REPLY* mechanism used to detect process failures and consensus. The mechanism gossips the information with a single process and spread the information to all alive processes using the same *PING REPLY* in the startup phase, growth phase, shrink phase and final phase. The proposed algorithm is separated into four logical tasks: *matrix initialization*, *detecting failures*, *merging the fault matrices*, and *check for consensus* according to MPI primitives.

In a large scale distributed systems, reaching an agreement among the processes is a fundamental need even in the presence of fault processes. A new epidemic approach i.e. information dissemination application [11] is proposed that simulate spreading process information and achieve global consensus in a decentralised fashion. Instead of using a separate application for disseminating the MPI processes information, the proposed work can be of use the same Gossiping approach in spreading process information. Katti *et al.* [12] introduced three algorithms based on epidemic protocols using xSim (extreme scale) Simulator for failure detection and consensus. In order to facilitate consensus detection, the first algorithm maintains an integer matrix at each process to store the status of all processes in the system. Thus the algorithm does not scale well.

This paper supplements the work in [12] and focus primarily on the first algorithm to increase its scalability. The proposed algorithm scalability has increased by maintaining the system view at every individual process in the form of a boolean matrix as shown in Figure 1. Consensus (global agreement) is achieved 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, say, ' $F_m$ ' holds  $n \times n$  elements where ' $n$ ' indicate number of processes. '0' indicate *process is alive*. '1' indicate *process have failed*. For instance, five processes ' $P_0$   $P_4$ ' are shown in Figure 1. We can check for consensus at any one of the alive process say ' $P_2$ '. We can detect a particular process say ' $P_1$ ' is alive or failed with other alive processes ' $P_0$ ,  $P_1$ ,  $P_3$ , and  $P_4$ ' by overlapping process ' $P_2$ ' with process ' $P_1$ ' which is shown in Figure 1 separately. Hence, in this case, consensus is detected by process ' $P_2$ ' for process ' $P_1$ '. The size of the fault matrix increases with the system size. To detect consensus, the number of Gossip cycles logarithmically increases with system size. The proposed algorithm is implemented and tested by means of MPI point-to-point communication primitives.

Snir *et al.* [13] proposed a programming model, MPI, increases the performance, execution speed and scalability. Enthused by this, MPI, a standard, defines a set of library methods that are useful to implement portable and scalable parallel applications. It is designed by researchers from software industry and academia to build large-scale parallel applications. MPI allow users to create parallel programs in C or Fortran 77 that run on parallel architectures more efficiently. MPI is a standard programming paradigm

created by the MPI forum, widely used on scalable parallel computers (SPCs). The major goal of MPI are scalability, portability, reliable messages transmission, added credibility to parallel computing, compatible on heterogeneous systems and achieving high performance. Many features that were included in MPI are as follows: Point to point communication, process groups, process topologies, profiling interface, collective operations, communication domains, environmental management and inquiry, bindings for C and fortran 77 languages. In the present work, out of all features available in MPI, point to point communication primitives were being used in implementing the proposed algorithm.

	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>
P <sub>0</sub>	0	1	0	0	0
P <sub>1</sub>	0	0	0	0	0
P <sub>2</sub>	0	1	0	1	0
P <sub>3</sub>	0	0	0	0	0
P <sub>4</sub>	0	1	0	0	0

1	1	1	1	1
---	---	---	---	---

Figure 1. Consensus detection on failure of process 'P<sub>1</sub>'

Margolin and Barak [14] presented a tree based fault tolerant algorithm using collective operations for parallel MPI applications that detect failures and failure recovery takes place as and when a failure has occurred. In [15], a user level failure mitigation (ULFM) specification is implemented in MPI to detect failures without stopping the applications execution in exascale systems. To solve the problem of studying and comparing different MPI fault tolerance techniques that helps in resuming system failures, Guo *et al.* [16] developed a bench mark suite: MATCH which will compare MPI fault tolerance designs for different scenarios. Chakraborty *et al.* [17] implemented a global restart model: EREINIT for bulk synchronous MPI applications to decrease failure recovery time and improve scalability. Failure detection, recover from failure and notification three basic mechanisms were optimized and implemented in MPI. Hassani *et al.* [18] designed a fault – aware MPI standard that implement fault – tolerant methods to confront the failures. A portable implementation of MPI tool, MPICH is introduced for developing parallel applications [19]. ULFM interface is used to program fault tolerant MPI in a large molecular dynamics application (a case study) [20]. [21] MPI is the best tool to implement parallelism in C, C++ and FORTRAN that has necessary standard libraries. Some studies has implemented the MPI according to their research problems. We have also implemented a unique MPI – based algorithm for our identified problem statement [14]–[21].

The paper is organized as: in section 2, details of the Gossip style matrix based failure detection and consensus algorithms are provided. Section 3 presents experimental results. Finally, section 4 concludes with presented work.

## 2. METHOD

Failure detection is essential for minimising failures and a core component of any resilience requiring infrastructure [22]. A failure detector is a distributed service capable of returning any processes and node's status, whether alive or dead. The overall performance of the high performance computing systems will affect in terms of latency to detect and propagate failures, and in terms of communication overhead and computation. In general, any process and node in the communication channel can communicate to any other process and node by sending messages that takes maximal time to be delivered. If a process and node has failed, then all the communication channels are emptied and is treated as permanent failure. Gossip approaches potentially incorporate random failure detection and propagation times [23]. Process and node randomly choose other processes and nodes with whom they share their failure information using gossip style protocols which is an alternative approach to implement scalable failure detectors. These protocols transmits information about all currently known failures using ping reply messages.

### 3.1. Randomised ping for failure detection

This section discusses randomised ping for failure detection as part of epidemic protocol. Failures are detected by a process by randomly ping other process periodically. Process  $p$  selects process  $q$  randomly to ping during  $T_{Gossip}$  cycle of length. Until the end of ongoing  $T_{Gossip}$  cycle, if process  $q$  replies, then process  $p$  detects process  $q$  as alive; or else failed. Figure 2 shows the algorithm pseudo code. During

$T_{Gossip}$  cycle, the chances to choose a process is 0 and 1 or more ping messages adhere to binomial distribution. Thus, the scalability of detecting a failed process by one or more alive processes have increased, thus achieving consensus by propagating the information. The implemented matrix based failure detection and consensus algorithm can permit medium-to-low delays and message losses, hence, it is completely fault tolerant.

---

**At each process  $p$**

---

**At every  $T_{Gossip}$  time (at each cycle)**

- 1 choose a random process  $q$
- 2 dispatch ping message to process  $q$

---

**At an event: a ping message received from  $q$**

- 3 dispatch reply message to process  $q$

---

**At an event: reply not received from  $q$  before timeout**

- 4 write process  $q$  has failed

---

Figure 2. Failure detection using randomised ping

## 2.2. Achieving consensus using global knowledge

This section discusses achieving consensus on MPI failed processes by enabling global consensus knowledge at every MPI process. Figure 3 shows the consensus algorithm where failures are detected by pinging random processes. The status of all the MPI processes is maintained in a boolean matrix  $F_i$ . An entry  $F_i[d, s]$  in the matrix denotes the status of process  $s$  as detected by process  $d$ . The consensus algorithm is separated into four logical sections as: i) initialisation, ii) detecting failures, iii) updating fault matrix and iv) check for consensus.

### 2.2.1. Initialisation

The line numbers 1 – 5 of Figure 3 assumes that every MPI process in the system is alive. No MPI process in the system detected any failures yet.

### 2.2.2. Failure detection

The line numbers 6 and 7 detects failures using randomised pinging by selecting a random process  $j$ , dispatching a ping message to process  $j$  piggybacking  $F_i$  fault matrix at every  $T_{Gossip}$  time. At line number 8, a timeout event is created during the current  $T_{Gossip}$  cycle to receive a reply message from process  $j$ . The line numbers 20 – 22 indicates a reply message is sent from process  $j$  piggybacking the fault matrix  $F_i$ . At line number 32, if no reply message is received by process  $i$  from process  $j$  at the end of the current  $T_{Gossip}$  cycle, process  $i$  detects (directly) process  $j$  to have failed.

### 2.2.3. Fault matrix update

The fault matrix is updated once a Gossip message (ping or reply) is received by process  $i$  from process  $q$ . The line numbers 23 – 27 and 29 – 31 updates the local fault matrix  $F_i$  by carrying out a logical OR operation between the corresponding elements in matrix  $F_q$  excluding  $i$ th row. At line number 28, an indirect local failure detection is performed by updating the process  $i$  in matrix  $F_i$  (row  $i$  in  $F_i$ ) to incorporate the process  $q$  detections (row  $q$  in  $F_q$ ). Sending entire fault matrix  $F_i$  information as part of Gossiping propagates process  $i$  detections along with other processes detection recognised by process  $i$ .

### 2.2.4. Check for consensus

Lastly the line numbers 9 – 19, consensus is checked on process  $s$  at  $i$  by carrying out a logical OR operation between  $s$ th column and its  $i$ th row corresponding elements. Thus, consensus has been reached when all alive processes in the system have detected a failed process.

---

**At each process  $i$**

---

$T_{Gossip}$  **cycle of length** and  
 $T_{out}$  **timeout period** are required

---

**Matrix initialization:**  
 //  $F_i[d, s]$ : process  $s$  status as detected by process  $d$   
 // Fault Matrix  $F_i[d, s]$  where  $0 \leq d, s < n$   
 1 for ( $d = 0, d < n, d++$ )  
 2 for ( $s = 0, s < n, s++$ )  
 3      $F_i[d, s] = 0$  // Every process in the system is alive  
 4   endfor  
 5 endfor

---

**At every  $T_{Gossip}$  time (at each  $T_{Gossip}$  cycle)**  
 // Failure Detection using randomised pinging  
 6     choose a random process  $j$   
 7     send a ping message to  $j$  piggybacking  $F_i$   
 8     receive a reply message from  $j$  by creating a timeout  
    event  $E_t = \langle \text{present cycle no} + T_{out}, j \rangle$   
 // check for consensus on process  $s$   
 9 for ( $s = 0, s < n, s++$ )  
 10    $temp = 0$   
 11 for ( $d = 0, d < n, d++$ )  
 12     if ( $F_i[d, s] \parallel F_i[i, d]$ )  
 13        $temp += 1$   
 14     endif  
 15 endfor  
 // a failed process is identified by all alive processes  
 16   if ( $temp == n$ )  
 17     consensus achieved on process  $s$   
 18     endif  
 19 endfor

---

**At an event: a message received from  $q$  piggybacked with  $F_q$**

---

20     if ( $message == ping$ )  
 21       reply message dispatched to process  $q$  piggybacking  $F_i$   
 22     endif  
 23 for ( $s = 0, s < n, s++$ )  
 24 for ( $d = 0, d < n, d++$ )  
 25     if ( $d \neq i$ ) //transmitting remote failure detection  
 26        $F_i[d, s] = F_i[d, s] \parallel F_q[d, s]$   
 27     else //indirect local failure detection  
 28        $F_i[i, s] = F_i[i, s] \parallel F_q[q, s]$   
 29     endif  
 30 endfor  
 31 endfor //merging the fault matrices

---

**At an event: no reply message received from  $j$  within  $T_{out}$  and timeout event  $E_t$**   
 // (direct Failure Detection) mark  $j$  to have failed  
 32  $F_i[i, j] = 1$

---

Figure 3. Failure consensus by global knowledge

### 3. RESULTS AND DISCUSSION

#### 3.1. Algorithm analysis

Epidemic protocols in large and extreme scale distributed systems are based on a peer-to-peer (P2P) model for computation and decentralised communication. In P2P networks, processes and nodes may join and leave the network arbitrarily (a.k.a *Node churn* [24]) in dynamic systems and may fail suddenly. It effects the robustness, efficiency of epidemic systems [25] and experimenting with a protocol during *node churn* is not an easy task. The proposed algorithm can be used to detect failures and achieve consensus using the MPI framework primitives. The implementation procedure for the proposed algorithm is given:

- a. Generate random processes and iterations
- b. Failures were injected before and during the execution of the algorithm
- c. Initialise fault matrix
- d. MPI\_MAX\_ERROR\_STRING is used for error handling of MPI
- e. MPI\_Comm\_size returns number of processes associated with a communicator
- f. MPI\_Comm\_rank returns rank of current process in the communicator
- g. MPI\_Barrier will synchronise between MPI processes
- h. MPI\_Gather collects local and global consensus information from all the processes
- i. MPI\_Type\_contiguous creates a new data type in contiguous memory locations for sending failure injection information
- j. MPI\_Type\_commit committing the new data type for communication
- k. MPI\_Send the failure injection information is sent to the destination
- l. MPI\_Recv the failure injection information is received from root
- m. MPI\_Irecv start / post reception for incoming ping / reply messages
- n. MPI\_Wtime returns an elapsed time in seconds
- o. MPI\_Test tests whether a send / receive operation is completed
- p. Displaying the total number of MPI processes that have reached consensus
- q. Producing the fault matrix after achieving global consensus

At every Gossip cycle, a ping reply communication is adopted in the algorithm in order to detect failures using randomised failure detection. During a Gossip cycle, a MPI process that receives ping messages follows binomial distribution. For this reason, the probability of receiving a single ping message/multiple ping messages by a failed process is very less. Therefore, at the earliest Gossip cycles, failed processes are detected.

Using the same two Gossip messages (ping reply) in a Gossip cycle, failure detected information is sent to all the alive processes across the system. Upon failure detection, the proposed algorithm achieved consensus logarithmically with system size. The failure information dissemination speed is doubled as there are two Gossip messages. Moreover, it is found that the failure information dissemination speed has increased and consensus time has reduced during the direct failure detection by individual process. Gossip messages required by the algorithm in each Gossip cycle is 2. As a result, the total number of gossip messages needed by the proposed algorithm is as:

$\text{Gossip messages needed at each process to detect consensus} = 2 * \text{Gossip cycles taken}$
--

The fault matrix is stored at each MPI process. Henceforth, the algorithm demand  $n^2$  memory units where  $n$  is total number of processes in the system. The algorithm is implemented using basic communication mechanism of MPIs “point to point communication operations” listed in section 3.1. The fault matrix is implemented as a boolean matrix to increase scalability of the algorithm. A process is excluded from further communication while simulating failure injection. The experiments were performed and tested on a single workstation desktop computer. The workstation system is running Ubuntu 18.04.1 LTS, openMPI 4.1.0 and gcc 7.4.0. Experiments were executed using openMPI to evaluate the proposed algorithm.

Failures were injected right before the execution and during the execution of the algorithm using epidemic protocols. Consensus is reached on failures when the epidemic protocols spread the information exponentially. The time out duration for one Gossip cycle length was set to 3 ms for  $2^5$  system size. The cycle length can be varied for a given system size which will permit to finish the matrix merge operations within the given cycle length. The varied gossip cycle length for different system sizes is required as the matrix merge operations take maximum cycle time. The scalability of the matrix and fault tolerance is tested through experiments. Pinging to the same node more than once is also allowed in the algorithm to use *redundancy* feature in case of Gossiping. Failures were injected randomly to the selected MPI processes while the proposed algorithm is run. The consensus is reached by each process at a different cycle number on the injected failures and so the total number of consensus reached by each process is recorded. The aforementioned facts have all been verified through experiments.

### 3.2. Results

Figure 4(a) shows the amount of gossip cycles consumed to achieve global consensus by each MPI process after single failure injection before the algorithm execution. Figure 4(b) shows the gossip cycles taken to reach global consensus during the algorithm execution. In both cases, it is observed that gossip

cycles count increases with system size in order to reach global consensus. Figure 5 shows the percentage of failures detected information spread at each MPI process after failures injected. Figure 6(a) and Figure 6(b) shows multiple failures injection at random cycles before and during the execution of the algorithm to reach global consensus. Eight failures were injected right before and during the proposed algorithm that satisfies the property of fault tolerance. It is also noticed that, gossip cycles count taken to reach/achieve global consensus increased to a slight extent. Table 1 displays the gossip cycle number at which each MPI process achieved consensus for a single failure injected before and after execution of the proposed algorithm. Table 1 displays the gossip cycle number at which each process achieved consensus for multiple '8' failure injected before and after execution of the proposed algorithm.

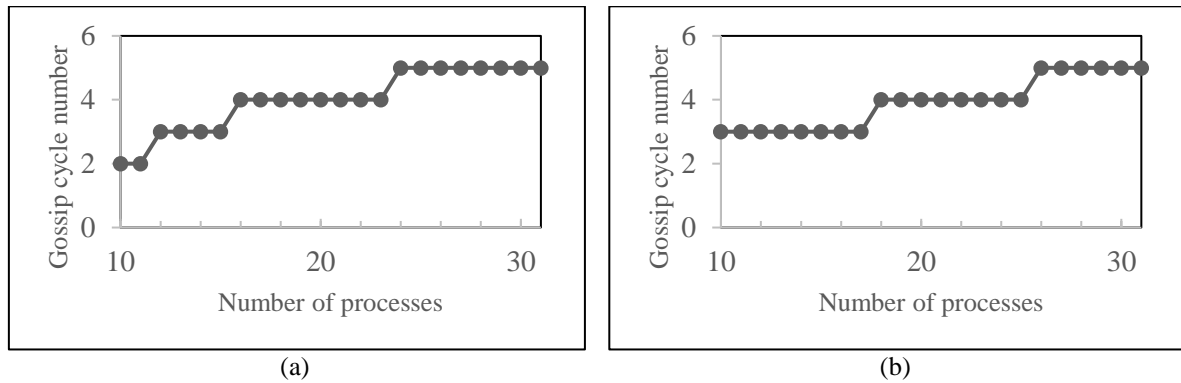


Figure 4. Cycle number at which each process has reached consensus after single failure injection (a) before algorithm execution and (b) during algorithm execution

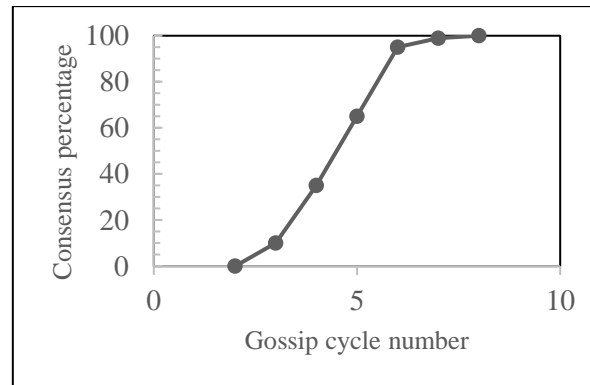


Figure 5. Consensus detected locally after failure injection

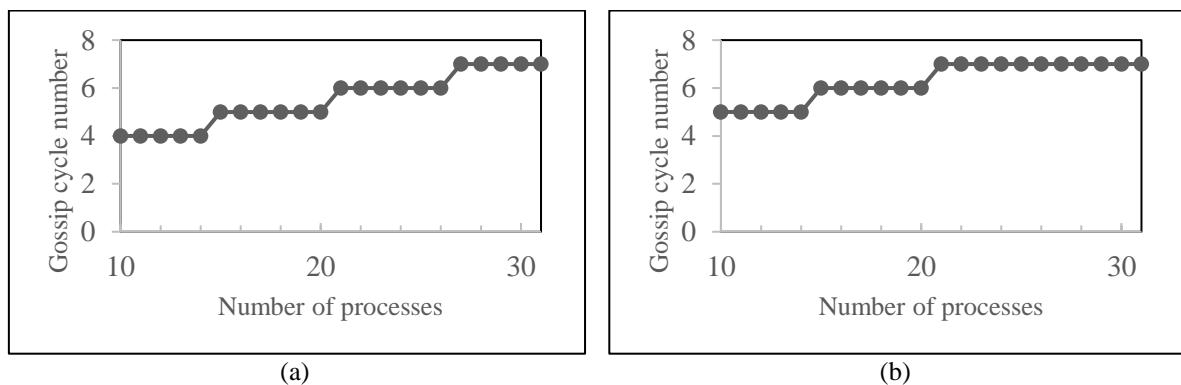


Figure 6. Cycle number at which each process has reached consensus after multiple '8' failures injection (a) before algorithm execution and (b) during algorithm execution

Table 1. The statistical analysis of consensus achievement

Gossip cycle #	Single failure		Gossip cycle #	Multiple failures	
	Before execution	After execution		Before execution	After execution
	Process #	Process #		Process #	Process #
2	10, 11	–	4	10–14	–
3	12–15	10–17	5	15–20	10–14
4	16–23	18–25	6	21–26	15–20
5	24–31	26–31	7	27–31	21–31

#### 4. CONCLUSION

This paper presented a Gossip style, matrix based failure detection and consensus algorithm that use randomised ping to detect MPI process failures. The algorithm is completely fault tolerant as it works even in the presence of single or multiple MPI process failures. Consensus is achieved on failed processes based on global knowledge: every process in the system maintains the status of all other processes. The proposed algorithm occupies more memory as every process uses a fault matrix of  $O(n^2)$ ,  $n$  indicate total processes count in the system. Failures were detected using a Gossip style protocol and disseminate them through out the system using the same Gossip messages. Experiments were tested on a workstation personal computer with  $2^n$  MPI processes. The scalability of the algorithm has improved by implementing with boolean values in the fault matrix at each MPI process.

#### REFERENCES





- [1] M. Snir *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014, doi: 10.1177/1094342014522573.
- [2] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996, doi: 10.1145/226643.226647.
- [3] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, 1993, doi: 10.1145/152610.152612.
- [4] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, April 1988, doi: 10.1145/42282.42283.
- [5] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed, Cambridge: Cambridge University Press, 2011.
- [6] A. Katti, G. D. Fatta, T. Naughton, and C. Engelmann, "Scalable and Fault Tolerant Failure Detection and Consensus" *EuroMPI '15: Proceedings of the 22nd European MPI Users' Group Meeting*, pp. 1–9, 2015, doi: 10.1145/2802658.2802660.
- [7] S. Ranganathan, A. D. George, R. W. Todd, and M. C. Chidester, "Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters," *Cluster Computing*, vol. 4, no. 3, pp. 197–209, 2001, doi: 10.1023/A:1011494323443.
- [8] K. Sistla, A. D. George, and R. W. Todd, "Experimental Analysis of a Gossip-Based Service for Scalable, Distributed Failure Detection and Consensus," *Cluster Computing*, vol. 6, no. 3, pp. 237–251, 2003, doi: 10.1023/A:1023592621046.
- [9] M. M. Ayiad and G. Di Fatta, "Agreement in Epidemic Data Aggregation," *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 738–746, doi: 10.1109/ICPADS.2017.00099.
- [10] A. Katti and D. J. Lilja, "Efficient and Fast Approximate Consensus with Epidemic Failure Detection at Extreme Scale," *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 267–272, doi: 10.1109/PDP2018.2018.00045.
- [11] M. Ayiad, A. Katti, and G. D. Fatta, "Agreement in Epidemic Information Dissemination," *International Conference on Internet and Distributed Computing Systems*, Sept 2016, pp. 95–106, doi: 10.1007/978-3-319-45940-0\_9.
- [12] A. Katti, G. D. 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, 2018 doi: 10.1177/1094342017690910.
- [13] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI-The Complete Reference*, vol 1, 2<sup>nd</sup> Ed, Cambridge: MIT Press 1998.
- [14] A. Margolin and A. Barak, "Tree-based fault-tolerant collective operations for MPI," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 14, 2020, doi: 10.1002/cpe.5826.
- [15] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteillera, and K. Teranishi, "Fault tolerance of MPI applications in exascale systems: The ULFM solution," *Future Generation Computer Systems*, vol. 106, pp. 467–481, 2020, doi: 10.1016/j.future.2020.01.026.
- [16] L. Guo, G. Georgakoudis, K. Parasyris, I. Laguna and D. Li, "MATCH: An MPI Fault Tolerance Benchmark Suite," *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 60–71, doi: 10.1109/IISWC50251.2020.00015.
- [17] S. Chakraborty *et al.*, "EReinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications," *Concurrent Computing*, 2018, doi: 10.1002/cpe.4863.
- [18] A. Hassani, A. Skjellum and R. Brightwell, "Design and Evaluation of FA-MPI, a Transactional Resilience Scheme for Non-blocking MPI," *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 750–755, doi: 10.1109/DSN.2014.78.
- [19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996, doi: 10.1016/0167-8191(96)00024-5.
- [20] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski, "Evaluating User-Level Fault Tolerance for MPI Applications," *EuroMPI/ASIA '14: Proceedings of the 21st European MPI Users' Group Meeting*, 2014, pp. 57–62, doi: 10.1145/2642769.2642775.
- [21] T. Rangunthar, P. Ashok, N. Gopinath, and M. Subashini, "A strong reinforcement parallel implementation of k-means algorithm using message passing interface," *Materials Today: Proceedings*, vol. 46, no. 9, pp. 3799–3802, 2021, doi:







- 10.1016/j.matpr.2021.02.032.
- [22] G. Bosilca *et al.*, “Failure Detection and Propagation in HPC systems,” *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 312-322, doi: 10.1109/SC.2016.26.
- [23] G. Bosilca *et al.*, “A Failure Detector for HPC Platforms,” *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, 2018, pp. 139–158, doi: 10.1177/1094342017711505.
- [24] P. Poonpakdee and G. D. Fatta, “Robust and efficient membership management in large-scale dynamic networks,” *Future Generation Computer Systems*, vol. 75, pp. 85–93, 2017, doi: 10.1016/j.future.2017.02.033.
- [25] M. M. Ayiad and G. D. Fatta, “An Adaptive Restart Mechanism for Continuous Epidemic Systems,” *Internet and Distributed Computing Systems (IDCS 2019), Lecture Notes in Computer Science, Springer*, vol. 11874, 2019, pp.57–68, doi: 10.1007/978-3-030-34914-1\_6.

## 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