The Journal of Supercomputing (2023) 79:9383–9408
https://doi.org/10.1007/s11227-022-05028-8

An unsupervised learning-guided multi-node failure-recovery model for distributed graph processing systems

Aradhita Mukherjee[1] · Rituparna Chaki[2] · Nabendu Chaki[1]

Accepted: 29 December 2022 / Published online: 13 January 2023

Abstract

Big data applications based on graphs need to be scalable enough for handling immense growth in size of graphs, efficiently. Scalable graph processing typically handles the high workload by increasing the number of computing nodes. However, this increases the chances of single or multiple node (multi-node) failures. Failures may occur during normal job execution, as well as during recovery. Most of the systems for failure detection either follow checkpoint-based recovery which has high computation cost, or follows replication that has high memory overhead. In this work, we have proposed an unsupervised learning-based failure-recovery scheme for graph processing systems that detects different kinds of failures and allows node recovery within a shorter amount of time. It has been able to provide enhanced performance as compared to traditional failure-recovery models with respect to simultaneous recovery from single and multi-node failures, memory overload and computational latency. Evaluating its performance on four benchmark datasets has reinforced its strength and makes the proposed model completely fit in with the status quo.

Keywords Distributed graph processing · Failure recovery · Hierarchical clustering

✉ Aradhita Mukherjee
aradhita.mukherjee.2016@gmail.com

Rituparna Chaki
rchaki@ieee.org

Nabendu Chaki
nabendu@ieee.org

[1] Department of Computer Science and Engineering, University of Calcutta, JD - II, Sector III, Salt Lake City, Calcutta 700106, India

[2] A. K. Choudhury School of Information Technology, University of Calcutta, JD - II, Sector III, Salt Lake City, Calcutta 700106, India

1 Introduction

Current research on big data analysis has shown the importance of the role of graphs. Social network analysis, medical diagnosis, and natural language processing are some big data applications where graphs are used to represent complex relationships and dependencies [1]. Such applications also use several machine learning and data mining (MLDM) algorithms [2, 3]. A parallel processing environment [4] is the natural choice for executing such complex algorithms where data size grows exponentially with time. In order to handle this huge load, graph-based parallel processing systems, continuously increase the number of compute nodes. However, the increasing number of compute nodes may lead to single or multi-node failures, either during job execution, or during recovery. This compels the entire system to stop its progress until the failed node recovers. Moreover, as the system overhead and computation time for failure recovery is high, subsequent node failures may further occur during the recovery of the initially failed node.

The failure recovery mechanisms used by GPSs can be classified into two broad categories: Checkpoint-based recovery and Replica-based recovery. In Checkpoint-based recovery, each node has to save its own information periodically on storage (like HDFS) [5]. After a failure occurs, each of the computing nodes reload their status from the latest checkpoint and redo all operations. The system recovers when all nodes have completed executing all operations that have been completed prior to the point of failure. The advantage of using a checkpoint-based recovery is that it can handle any type of failures. However, it needs a huge amount of persistent storage to store states of each node in the system as checkpoints. Recovery time is also high for such systems. Some of the recent GPSs like Pregel [6], PowerGraph [7] and Distributed GraphLab [8] follow checkpoint-based recovery mechanism [9]. Among them, Power Graph has been widely used because of its faster graph-processing rate and higher scalability though it suffers from high communication complexity [10]. Moreover, the messages the node receives during recomputation might not always arrive in the same order as they do during regular execution. This enforces an additional constraint for the system to be insensitive to message ordering [11].

A replica-based recovery system, on the other hand, works by creating a replica of each node. This is achieved by local memory access which, in turn, decreases computational latency since atleast $k + 1$ replicas are required to make the system recover from k failures [2]. Maintaining replicas, also has a high storage overhead. The master vertex and its replicas, here, are synchronized through message passing.

In general, GPSs partition graphs into several sub-graphs which are then placed in individual computing nodes. Graph partitioning approaches like edge-cut and vertex-cut are often used for this purpose. After a failure, the sub-graphs that reside in a failed node are redistributed over multiple healthy nodes. This leads to reduction in computation cost but increase in communication cost [12]. Nevertheless, a good partitioning algorithm that balances this trade-off, is lacking. The storage and computational complexity of the two failure recovery mechanisms discussed above, also motivates the need for development of a robust and optimal fault-recovery GPS that would allow recovery from both single and multi-node failures.

In this work, we have proposed a scalable failure-recovery mechanism for distributed GPSs. GPS plays an important role in community applications. Community applications need global data pool where people can share their data. Data sharing is the sole purpose of such applications. Many variants of such applications demand that the data be encrypted during communication. Centrally controlled social application service providers also use a global data pool model. However, such applications require central authorization and thus suffers from issues like single point of failure and scalability. Hence, decentralization has been adapted as an alternative by most of the open source communities. One such decentralized application Dat [13], now known as, Hypercore [14] is used to share large amount of scientific datasets between research groups [15]. Secure-Scuttlebutt (SSB) [16] is a similar social application that is mainly used for blogging and code development, while Ledger-Mail [17, 18] is another decentralised email transfer system. These type of social applications are only interested in a subset of the global data pool. Thus, it is feasible to locally store the data. Such applications use replicated, authenticated, single-writer, append-only log files which consist of event chains for each participant. Such log files are replicated through gossip algorithms to produce an eventually-consistent social application. These types of applications are best suited for small and trusted groups [19].

The proposed model has been driven by the single node-single user principle, where each user/participant has been considered as a single node with its own storage space. Coordination between participants has been maintained using append-only data structures. A clustering algorithm has been then used to partition the graph of users/participants into several virtual clusters. Each virtual cluster has a cluster head, also known as imitator. Every node maintains a log file, which is divided into some chunks. The number of chunks has been decided based on the number of clusters obtained in the earlier step. The encrypted chunks of the log files are sent to the imitator node of each cluster in a round-robin fashion (in ascending order of cluster ID), which are accessed only by the owner node during the recovery period. The imitator nodes keep track of the actual location of the chunks. Each node periodically sends a heartbeat to its neighbours present in its own cluster. An absence of heartbeat from a node for a long period of time, indicates that the node is dead. In order to recover a failed node, a broadcast message is sent to all the virtual clusters. On receipt of this message, each imitator retrieves the backup files from the nodes in its own cluster and sends them to the failed node. The node recovers itself upon receipt of backup files from all clusters. The proposed failure-recovery model, when evaluated on four benchmark datasets, has shown enhanced performance, enabling simultaneous recovery from both single and multi-node failures within shorter amounts of time with low memory overload as compared to traditional GPSs.

2 Related work

In order to handle the dynamic increase in size of data, a majority of GPSs increase the number of computing nodes, which in turn, increases the probability of node failures. In checkpoint-based recovery models, each node saves its state periodically

in storage, which is reloaded by the failed node during its recovery period [20]. Periodic storing and reloading from storage during failures degrade performance and incur additional overhead since it not only involves reloading the checkpoint, but also implies recomputation and message passing among all compute nodes. Based on the recent checkpoint, it re-executes missing computation of all nodes residing in both healthy and failed nodes. A subsequent failure during recovery of the node, implies rolling back each compute node and restarting the recovery method from scratch [5].

The failure recovery model proposed in [11] is an improvement over conventional checkpoint-based recovery where the authors have removed the overhead due to high recomputation cost for the sub graphs located in the healthy nodes, since failures only affect a small fraction of compute nodes. It should be noted that a healthy node's subgraph can contain both its original subgraph (whose computation is never lost) and a set of recently received partitions (whose computation is partially recovered) as a result of prior failures. The authors have also split up the recomputation duties for the subgraphs in the failing nodes over several compute nodes to increase parallelism.

An alternate approach for node recovery [9] combines global check pointing with local logging. Once a failure is detected, all remaining vertices are requested to send messages to the failed vertices. In this scheme, each node maintains a local log for outgoing messages. Hence, no recomputation is needed. A column-wise message compression method is used to reduce the logging overhead. The sub-graphs present in the failed nodes are re-partitioned and distributed over healthy compute nodes for the sake of recomputation towards parallel recovery. After recomputation, partitions are again redistributed among all the compute nodes thereby increasing performance overhead. The problem associated with this recovery scheme is synchronization between local storage and global storage, which, if not done properly, may lead to data inconsistency. The storage overhead associated with this kind of recovery is also quite high as compared to checkpoint-based recovery methods since both local and global storage's have to be maintained.

Phoenix [21] is a distributed-memory application that has been developed for graph analytics. It serves as a substitute for check-pointing which resets the entire calculation to its state just prior to the fault. Phoenix reloads graph partitions from stable storage on the revived hosts that take the place of the failed hosts whenever a fail-stop error has been identified. If any of the nodes on these hosts have proxies on surviving or healthy hosts, it may be possible to recover the states of those proxies. It is significantly superior to checkpoints and can handle any kind of failures. However, memory overhead remains a problem since the proxy nodes live within revived hosts since it needs to keep stable storage on those hosts. Additionally, because synchronisation is required both during regular execution and during the recovery mechanism, this method entails synchronisation overhead.

In another recent investigation [2], the authors have employed vertex replication to handle node failures. Following this scheme, atleast $k + 1$ replicas need to be maintained in order to handle k machine failures. This incurs huge storage overhead for storing $k + 1$ replicas. There is a imitator node which handles replica management. The imitator node creates auxiliary replicas for vertices without replication

and synchronizes the full states of a master vertex to its replicas through message passing. The locations of the replicas are chosen randomly by the imitator to create a fault recovery system. The authors have proposed two approaches for node recovery - Rebirth-based recovery and Migration-based recovery. In Rebirth-based recovery, if a node that has crashed, contains any replica vertex, that vertex would be recovered by the master vertex while, in Migration-based recovery, if a node that has crashed, contains master vertex then the mirror vertex would be promoted as a master vertex. This recovery scheme suffers from huge space overhead since it involves storage of mirrors and replicas. Besides, synchronization between master replica and mirror replica is expensive as they reside at different sites. Maintaining data consistency between them is also difficult.

Some recent fault recovery protocols do not perform partitioning of graphs for failure recovery, rather, they believe in single node-single user architecture. In SSB [16, 22], every single user is considered as a single node. Thus, there is no need for graph partitioning. This decentralized, peer to peer protocol does not need run-time configuration checking, which makes it more efficient. Here, each user in a graph stores its data in a log file. This increases integrity of the received data. However, usage of gossip-based replication protocols makes them suffer from eclipse attacks [23].

In [24], a single node failure recovery method is proposed where each vertex or user has been considered as a single node with its own storage space. Here, data within a log file of a node has been divided into chunks and distributed to $n-1$ neighbouring nodes. When a node crashes, it receives its chunks from its $(n-1)$ neighbours. However, the recovery model in [24] is constrained by the fact that it only focuses on single-node failures. It does not provide any solution when multiple node failures occur simultaneously. In situations where recovery of a node takes long, a further node failure may occur within that duration. This may create an endless recovery loop and the model proposed in [24] has been inadequate in handling such simultaneous failures.

3 Problem statement

On analyzing the existing state-of-the-art methods for failure recovery in GPSs, we have arrived at the conclusion that a recovery mechanism for single and multi-node failures that is scalable and less expensive in terms of recovery latency, is lacking at present. Most of the recovery approaches consider a server as a compute node (N) that contains more than one partition (P), which implies that a node crash would result in the crash of all vertices within a partition $P_i = (V_i, E_i)$, where $V_i \subseteq V$ and $E_i = (v_i, v_j) \in E | v_i \in V_i$, either because the recovery method suffers from a high computational cost as well as high communication costs, or involves high communication overhead. In general, the computational cost ($T_p[i]$) is measured as $T_p[i] = \max_{n \in N} \sum_{T(v,i)} \{v \in A(i) | \boldsymbol{\Phi}_{pi}(\boldsymbol{\Psi}(v)) = N\}$, where $T(v, i)$ denotes the computation time of v in the normal execution of superstep i. Let $A(i)$ be the set of vertices that perform computation and $\boldsymbol{\Phi}$ be the vertex-partition mapping, and $\boldsymbol{\Psi}$ be the mapping between a failed partition to a healthy compute node. Then, the communication

cost is measured as $T_m[i] = \sum_{\mu(m)/B} \{m \in M(i) | \Phi_{pi}(\Psi(m.u))) \neq \Phi(\Psi(m.v))\}$, where $M(i)$ is the set of messages forwarded when re-executing superstep i, $m.u$, $m.v$ and $\mu(m)$ are the source vertex, destination vertex and size of message m respectively, and B is the network bandwidth [11].

However, the present state-of-the-art methods for failure recovery involve reassignment and recomputation during recovery, which heavily influences the total recovery cost that includes both computational and communication cost. Hence, the need to develop a scalable fault recovery mechanism that would enable recovery from both single and multi-node failures, while optimizing both communication and computational cost [min $\sum (T_p[i] + T_m[i])$], has thus become indispensable.

4 Prerequisites

4.1 Partial replication

Data replication [25, 26] plays an important role in distributed system as it ensures data availability. Full replication increases availability by replicating the entire database at every site. However, it makes update operations slower and may lead to inconsistencies. In partial replication, each site holds subsets of data in order to increase scalability. During execution of a transaction, all data items are not available at a single site. Thus, to ensure consistent data delivery, inter-site synchronization is required. Message passing is used for communication among sites.

4.2 Distributed graph processing

Graphs represent relationships between data items. A graph consists of (v, e), where v represents actor, user or node and e represents edge or relation between the nodes. As the number of nodes grows, graphs tend to become larger and sparse. A social network with billions of users, web access history, or an online game network demands a graph like structure to represent their interconnections. Such bulk amount of information cannot be stored within a single computing node. To enable parallel processing, graphs are sometimes partitioned into clusters, each of which may then be distributed over several computing nodes. Several data mining algorithms uses distributed graph processing techniques for analysing information. However, such algorithms demand high parallel computation, efficient data partitioning, and communication management mechanisms. Recent models implementing aforesaid high-level programming abstractions include vertex-centric models and neighbourhood-centric models [27].

4.3 Agglomerative hierarchical clustering

Hierarchical clustering [28, 29] is an unsupervised clustering algorithm which organizes similar objects into groups. At the end of this hierarchy we get a set of distinct clusters. It can be of two types: agglomerative or divisive. An

agglomerative hierarchical clustering uses a bottom up approach to group a set of objects into one cluster based on similarity. It is a greedy algorithm, where initially each object is treated as an individual cluster. Subsequently, two distinct clusters are merged together based on some selection criteria like single-linkage, complete-linkage, average-linkage or centroid-linkage. This process is continued until we arrive at one single cluster containing all objects. The clustering hierarchy is often represented by a dendrogram.

5 Proposed methodology

In this work, we have proposed a failure recovery model for both single and multi-node failures in a distributed graph processing system. Here, every single user has been considered as a single node. Initially, we have partitioned the graph into multiple clusters. Each user maintains a log file to locally store their outgoing messages. Log-file of each user has been divided into some partitions based on the number of clusters and sent to their neighbouring clusters for backup. When a node failure occurs, the failure is detected by the neighbouring nodes of the failed node and they refrain from sending further messages to the failed node. When the failed node wishes to recover, it broadcasts a message to all neighbouring clusters. Subsequently, the neighbouring clusters send the compressed data to the failed nodes and it recovers.

The proposed work has been divided into three phases:

1. Phase I or Partitioning: Partitioning the nodes in the graph into several clusters.
2. Phase II or Chunk distribution: Splitting the log files of each node and distributing them to other virtual clusters
3. Phase III or Failure recovery: Recovery of failed nodes.

A schematic diagram of the proposed failure recovery management scheme has been depicted in Fig. 1. The assumptions used in this work have been defined below:

Definition 1 (*File management*) Each node within a virtual cluster maintains its own log file for communication. Some nodes may have an extra file known as a backup file which stores chunks sent by neighbouring cluster. Backup files have been used during the recovery process. The number of backup files (γ_i) within a node, is decided by the imitator (cluster-head) and depends on the number of neighbouring clusters.

Definition 2 (*Communication system*) Inter- and Intra-communication among the virtual clusters obey peer-to-peer distributed network protocol using a push-pull pattern through the TCP channel. Here, a sender node 'push' es message into the channel and the receiver receives the message from the channel through 'pull'.

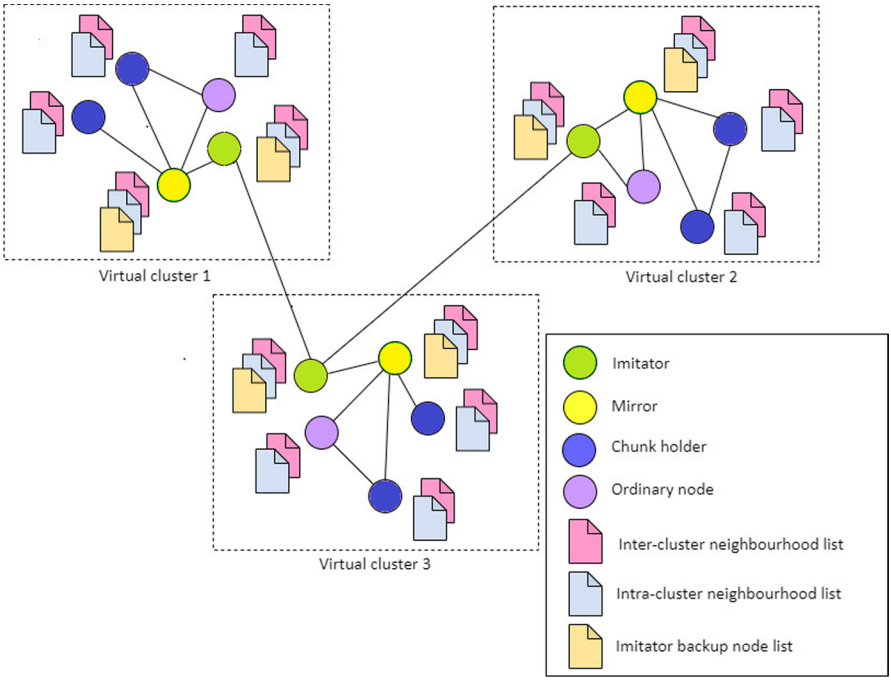


Fig. 1 Proposed framework for failure recovery management

Definition 3 (*Network channel*) During recovery, a failed node sends a broadcast message to other clusters through connection-less protocol (UDP) which decreases the latency period. The other data transfer operations of nodes have been done using transmission control protocol (TCP), which increases stability and reliability of the overall networking process.

Definition 4 (*Message types*) Several types of messages have been used in this work. When a node initiates, it sends a 'READY' message to all its neighbouring nodes in order to indicate that it is ready for communication. The 'READY' message contains an '*Alive*' signal, *cluster_id* (cluster identifier) and *self_id* (unique IP address). If a node fails, it tries to initiate recovery on being live again. During recovery, a failed node broadcasts a 'LOST' message to all neighbouring virtual clusters. The nodes in the virtual clusters with backup can help the lost node to recover itself. The 'LOST' message contains a '*Lost*' signal, *cluster_id* (cluster identifier) and *self_id* (unique IP address).

Definition 5 (*Node state detection*) To check whether a node is dead or alive, a heartbeat signal is generally used. Here '*Alive*' Signal has been used as the heartbeat of a node. A node that is not able to generate an '*Alive*' signal for a long time, is considered a dead node.

Definition 6 (*Inter-cluster_neighbourhood_list*) This neighbourhood list is a tuple containing *self_id* (unique IP address), *neighbour_list* (containing IP addresses of nodes in neighbouring clusters). This list is present with every node, including the imitator node.

Definition 7 (*Intra-cluster_neighbourhood_list*) This neighbourhood list is a tuple containing *self_id* (unique IP address), *neighbour_list* (containing IP addresses of nodes in its own cluster) and *uid* (universal port address). An universal port has been used to broadcast message(s) and listen to broadcast-ed message(s). This list is present with every node, including the imitator node.

Definition 8 (*Imitator_backup_node_list*) This list contains three tuples *sender_id* (unique IP address of sender node from another cluster), *cluster_id* (cluster identifier of the sender node) and a *backup_node_list* (containing IP addresses of two nodes within its own cluster who would hold chunks coming from outside).

5.1 Phase I: partitioning

In this phase, we have partitioned the set of nodes into several clusters. A graph partitioning problem can be defined as dividing the graph into two sets A and B such that weight of edges connecting vertices in A to vertices in B is minimum, and size of A and B are similar. This is an NP-hard problem. A graph edge cut or vertex cut has been the de facto method to partition a graph. If a fraction of nodes and edges are closely connected to one another, they are recognized as one community (or one partition), otherwise, they are considered as multiple communities. Hence, the network is partitioned such that nodes within a single partition have maximum number of edges between them while nodes in different partitions are loosely separated with minimum number of edges between them. This is similar to the objective function of a clustering algorithm where we aim to maximize intra-cluster similarity and minimize inter-class similarity. This is why we have employed an agglomerative hierarchical clustering algorithm to create the initial partitioning. Typically, a clustering algorithm is unsupervised since it does not use ground truth labels. The input to the clustering algorithm is the adjacency matrix $W(i, j)$ representing the distances between the nodes i and j where $i \in n$ and $j \in n$ (where n is total number of nodes in a graph) calculated using Breadth First Search (BFS). The optimal number of clusters has been decided using Silhouette score [30] and Calinski-Harabasz score [31]. Each virtual cluster thus produced, contains a set of users/nodes. The vertex-to-virtual cluster mapping $f : u_i \longrightarrow V_i, u_i \in U$, $V_i \in V$, where U is the set of users and V is the set of virtual clusters, stores information about which user belongs to which virtual cluster. Algorithm 1 explains the working of the partitioning phase.

Algorithm 1: Creating virtual clusters

Input : Input: Graph $G(v, e)$
Output: Set of virtual clusters V
$D \leftarrow$ Calculate distances using $BFS(G(v, e))$
$k \leftarrow$ Find optimal number of clusters using $Silhouette\_score()$ and $Calinski\_harabasz\_score()$
$linkage \leftarrow$ 'average'
$V \leftarrow Agglomerative\_Hierarchical\_Clustering(D, k, linkage)$

5.2 Phase II: chunk distribution

Once the virtual clusters are created, each node passes through several states: Initial, Ready, Active and Failed. At the Initial state, each node creates its own directory, log files and backup files. Every node in the network maintains an Inter-cluster_neighbourhood_list and an Intra-cluster_neighbourhood_list, while an imitator node contains the above two lists along with the Imitator_backup_node_list. When a node is in Ready state, it tries to communicate with other nodes. It then broadcasts a 'READY' message. On receipt of the 'READY' message, the connection is established. Subsequently, all sub-processes like communication, file management and node state detection are initiated and the sender node moves into an Active state. Throughout the period of message communication, a node remains in Active state. It sends periodical heartbeats following a Publisher-Subscriber (pub-sub) manner, to its neighbours. The node state detection process runs in the background and checks if a node is running or dead. Once a node is detected as dead, we conclude that the node has failed. We call it the Failed state. The connection is hence terminated and closed. Apart from these, there are two other sub-processes like chunk distribution and chunk collection. Chunk distribution is executed when a node is in Active state. On detection of failure of a node, the chunk collection method is run.

During chunk distribution process, for each cluster, the node with maximum in-degree, is selected as an imitator. Within every virtual cluster, the imitator node also has a mirror copy which is promoted as the imitator if the former fails. After the imitator is selected, the log file of each node within a virtual cluster is divided into $k - 1$ chunks (where k is the number of virtual clusters), and sent to $k - 1$ neighbouring clusters, in a round-robin fashion. This process is repeated at each interval of time. The complexity for the same is $O(n)$ (See Theorem 1). The above steps have been drafted in Algorithm 2.

Algorithm 2: Chunk distribution

```
Input  : Input: Set of virtual clusters V, Number of clusters k
Output: Distribution of chunks
/* Select imitator node for each cluster                          */
max ← 0
for each cluster V_i ∈ V do
    for each node u_i ∈ V_i do
        if indegree(u_i) > max then
            max ← indegree(u_i)
            im ← i
        end
    end
    Imitator(V_i) ← u_im
    max ← 0
end
/* indegree(x) finds the degree of node x                         */
/* Imitator(V_i) holds imitator of a particular cluster           */
for each cluster V_i ∈ V do
    for each node u_i ∈ V_i do
        Parts(u_i) ← Divide logfiles of u_i into (k − 1) partitions
        for each data chunk Chunk_j ∈ Parts(u_i) do
            V_k ← Chunk_j where k! = i
        end
    end
end
```

On receiving a chunk from a node in a neighbouring cluster, the imitator node distributes the chunk to two nodes within its own cluster. These two nodes are chosen randomly by the imitator. The reason for choosing two nodes is to restore backup if anyone of them fails. To keep track of chunk locations, the imitator node maintains an Imitator_backup_node_list. The above discussed functions of an Imitator have been depicted in Algorithm 3. This mechanism of storing and retrieving backups ensures recovery from multiple-node failures.

Algorithm 3: Chunk backup by Imitator

```
Input  : Data chunk C from each node
p1, p2 ← Randomly select two nodes from its own cluster.
p1, p2 ← C          // Distribute chunks to two backup nodes p1 and p2
Update Imitator_backup_node_list
```

5.3 Phase III: failure recovery

Once a faulty node is detected by its neighbours, no further message communication takes place with the faulty node. On being live again, when the failed node tries to recover itself, it broadcasts a 'LOST' message to its neighbouring virtual clusters available in the Inter-cluster_neighbourhood_list. On receipt of the 'LOST' message, the imitator nodes of the neighbouring clusters searches the *sender_id*

within its Imitator_backup_node_list. If the *sender_id* is present, it retrieves the corresponding backup files from the *backup_node_list*. If both the backup nodes are alive, it fetches data from any one of them. The data is then compressed and uni-cast to the failed node. On receipt of data from all the neighbouring clusters, a failed node recovers into a safe state.

On receiving the backup files from the neighbouring clusters, it identifies its own cluster and rejoins its cluster. In order to recover further into the current state, it then requests other nodes within its own cluster to send information regarding latest communication. The recovered node then moves into Active state once more. It has been observed that multiple nodes within the network, if failed simultaneously, would recover concurrently since chunks of backup files are distributed on different clusters and the same chunk, within a cluster, is replicated twice. The entire recovery mechanism has been explained in Algorithm 4.

Algorithm 4: Algorithm for node recovery

```
Failed node creates a directory and an empty log file using its self_id.
Failed node broadcasts a 'LOST' message through uid to k − 1 clusters.
Each cluster imitator receives a 'LOST' message and checks
if self_id ∈ sender_id then
    p1, p1 ← Search backup_node_list for nodes with backup
    if status[p1] = 'Alive' and status[p2]! = 'Alive' then
        Transmit data chunk present in p1 to failed node
    else if status[p1]]! = 'Alive' and status[p2] = 'Alive' then
        Transmit data chunk present in p2 to failed node
    else
        Transmit data chunk present in p1 or p2 to failed node
    end
end
Failed node recovers by receiving backup from all clusters.
Recovered node now resumes communication.
```

6 Experimental setup

6.1 Datasets used

The proposed failure recovery method has been evaluated on four benchmark datasets. Datasets 1 and 2 have been derived from two email datasets from two departments of an European research institution [32]. These datasets consist of nodes and edges where a directed edge between two users *A* and *B* exists if *A* has sent an email to *B*. We have assumed all links between nodes to be stable. Dataset 3 has been derived from a Facebook 'circles' (or 'friends lists') data collected through user survey [33]. Dataset 4 has been derived from LiveJournal which is a social networking and journal service that allows users to create and share blogs, journals, and diaries. It has over 4 million vertices (users) and approximately 70 million directed edges (associations between users). [34]. The number of nodes and edges for all four datasets have been shown in Table 1.

Table 1 Dataset description

Dataset#	#Nodes	#Edges
Dataset 1	162	1772
Dataset 2	309	3031
Dataset 3	4039	88234
Dataset 4	3, 997, 962	34, 681, 189

6.2 Hardware and software setup

We have used the institutional server to carry out the experiments in this study. This server is equipped with one PowerEdge R740/R740XD Motherboard, two Intel Xeon Silver 4216 2.1G processors, two 22M Cache, four 32GB RDIMM, 2933MT/s, Dual Rank, four 10TB 7.2K RPM NLSAS 12Gbps 512e 3.5 in Hot-plug Hard Drive and two 480GB SSD. This work has been implemented using Python 3.7.7 on the CentOS 7 Operating System.

6.3 Environmental setup

In this scope of work, we have considered each node as an individual user. At the onset, the dataset has been partitioned into a set of k virtual clusters. Within each cluster, the node having the maximum in-degree has been considered as the imitator. Every node within a cluster creates two log files: the *self_log* file and the *backup_log* file. The *self_log* file stores logs (conversation messages) for the node itself, while the *backup_log* file is created in order to store the backup received by the imitator from neighbouring clusters.

Periodically, the content of *self_log* file is divided into $k-1$ partitions, encrypted and sent to $k-1$ neighbouring clusters in a round robin manner. Chunks contain *node_id* and data in ordered fashion. After being received by the imitator of a neighboring cluster, these chunks are stored within the *backup_log* file of a node selected by the imitator of that cluster. *backup_log* files are append-only log files and they have been named after the node whose backup it contains.

The communication among virtual clusters obey peer-to-peer network topology which has been simulated using virtual local area network (VLAN). UDP has been used to broadcast signals in order to provide high-speed connection-less communication. For data transfer, a stable connection is required, which has been established through TCP. A producer-consumer mode of communication has been followed here. The producer node pushes data into the communication channel and receiver pulls data from the communication channel. Message generation follows pub-sub pattern.

Nodes send 'READY' messages containing '*Alive*' signal to initiate communication. When no heartbeat is received from a node for a long period, it is assumed that the node has failed. The recovery of the failed node follows the steps described in Sect. 5.3.

7 Proof of correctness

Theorem 1 *The complexity of chunk distribution is $O(n)$, where n is the total number of nodes in the network.*

Proof If there are k virtual clusters, the log file of each node is divided into $k-1$ chunks and distributed to all neighbouring clusters. If chunk division and distribution take constant amount of time $c1$ and $c2$ respectively, then, the total time taken is equal to $(k-1)\times c1+(k-1)\times c2$. For n nodes in the network, total complexity is thus $[(k-1)\times c1+(k-1)\times c2]\times n = n\times(k-1)\times(c1+c2)\approx O(n)$ since $k << n$. □

Theorem 2 *The complexity of failure recovery of a node is $O(k+m)$, where k is the number of clusters and m is the number of nodes within a virtual cluster and k, $m << n$ where n is the total number of nodes in the network.*

Proof Recovery of a node is a two-step process. Initially, a failed node receives backup from imitators of its neighbouring clusters and reaches a safe state. Thereafter, it receives backup of latest state information from the nodes within its own cluster and reaches the current state. Complexity for the former step is $O(k-1)$, where $k << n$, since there are at most $k-1$ neighbour clusters of a node. Considering $m-1$ neighbours of a node within its own cluster, the latter step would have a complexity of $O(m)$, where $m << n$. Thus, total complexity becomes $O(k+m)$. □

Theorem 3 *Multiple node failure recovery can be made concurrently.*

Proof In this work, the log file of each node is distributed to $k-1$ clusters. Within each virtual cluster, the imitator randomly selects two nodes who would store the backup chunks received from a neighbouring cluster. The imitator then keeps track of the same using its *Imitator_backup_node_list*. The reason for selecting two backup nodes is to ensure recovery of the failed node even if one of the backup nodes fail. In a non-clustered environment, the recovery of a node would depend on all its neighbour nodes, many of which may also fail simultaneously. This would prevent recovery of multiple nodes at the same time. In the proposed method, we have initially partitioned the nodes in the network into several virtual clusters. A clustered environment ensures that the recovery of a failed node depend on its neighbours not from its own cluster but from a neighbouring cluster. Thus, distributing backup files into multiple clusters ensures better availability. Hence, two failed nodes from different clusters can be recovered simultaneously. However, Theorem 3 does not hold if both backup nodes fail simultaneously. □

Theorem 4 *The Chunk distribution algorithm has a space complexity of $O(s\times m\times k)$*

Proof The space complexity is determined by the log file size (s), the number of clusters (k), and the number of nodes within each cluster (m). In the proposed method, each node is considered as an individual user and contains its own log file. We assume that the size of the log file for each node is s. Each node divides its log file into $(k-1)$ chunks. As a result, the space consumption of each chunk is $s/(k-1)$. Each of these $s/(k-1)$ chunks are distributed to each of the $(k-1)$ virtual clusters. Again, each cluster contains m nodes. Therefore, each imitator node in a cluster receives chunks from all m nodes in $(k-1)$ neighboring clusters. The memory consumption by each Imitator node is thus $s/(k-1) \times m \times (k-1) = s \times m$. The overall memory consumption for k imitator nodes is thus $s \times m \times k = O(s \times m \times k)$. □

Theorem 5 *The node recovery algorithm has a space complexity of* $O(s)$.

Proof When the failed node tries to recover itself, it broadcasts a 'LOST' message. On receipt of the 'LOST' message, the imitator nodes of the $(k-1)$ clusters search the *sender_id* within its *Imitator_backup_node_list*. If the sender_id is present, it retrieves the corresponding backup files from the backup node list. The chunk (having size $s/(k-1)$) is then compressed and unicast to the failed node with size. On receipt of data from all the $(k-1)$ clusters, a failed node recovers into a safe state. The memory consumption by the failed node is thus $s/(k-1) \times (k-1) = O(s)$. In order to progress it then requests that other nodes within its own cluster send information about the most recent communication. The space consumption for intra-cluster communication can be considered insignificant in comparison to inter-cluster communication. □

Assertion 1 The proposed failure recovery method shows good synchronization capability which preserves the content of the failed node making it consistent.

Proof The proposed failure recovery method shows good synchronization capability which preserves the content of the failed node making it consistent. In the proposed mechanism, the content of the log file is divided into $(k-1)$ parts, which are encrypted at the time of sending with the original line number and *node_id*. This encrypted message is then sent in round-robin fashion to a $(k-1)$clusters following a push-pull pattern. At the receiving end, the messages are decrypted and stored in the receiving cluster based on line numbers.

Once a node fails, it sends a "LOST" message to the $(k-1)$ clusters, which receive and decrypt it to determine if the *node_id* is present in the *neighbour_list*. If the *node_id* is present in the *neighbour_list*, then the node unicasts the data present in the log file (with line numbers embedded in it) of the failed node received earlier, to the failed node. The failed node then restores to its current configuration. After receiving backup files from every other cluster, it will reestablish all the connections and successfully recover to its previous full-fledged working condition.

The presence of line numbers in the log files received as backup ensures synchronization. Further, usage of the push-pull method ensures message synchronization and reduces communication overhead [35]. □

Assertion 2 The communication overhead of the proposed method is $O(k)$.

Proof We have used the gossip protocol in the proposed communication method. The expected number of gossip messages is $O(mlogd)$ where m is the number of clusters and d is the number of nodes that the initiator sends a message to [36]. In the proposed method, a failed node sends chunks to $(k-1)$ clusters and the Imitator node in each cluster stores the received message in any two nodes at random, by linearity of expectation, the total expected number of messages is$O((k-1) * log2) = O(k)$. □

8 Performance evaluation

The proposed failure recovery model has been evaluated on four benchmark datasets. The initial part of the proposed method involves a base clustering step where, for each dataset, the nodes have been partitioned into several clusters using an agglomerative hierarchical clustering algorithm. We have used two internal validity indices Silhouette coefficient [30] and Calinski-Harabasz index [31] to compute the number of optimal clusters. The Silhouette score or Silhouette coefficient measures the intra-cluster similarity of an observation (cohesion) compared to the inter-cluster dissimilarity (separation). It ranges from -1 to $+1$. The higher the value, the better is the clustering solution. The Calinski-Harabasz index (also known as Variance Ratio Criterion) is another measure to evaluate a clustering solution where the within-cluster variance is compared to the between-cluster variance. The higher the score, the better is the clustering performance. To find the optimal value of k (number of clusters), we have varied its value from $2-10$ and selected the k value on which both Silhoette score and Calinski-Harabasz score agreed upon. The optimal number of clusters for datasets 1, 2 and 3 have been 7, 9 and 8 respectively. For the fourth dataset, the optimal number of clusters has been considered as 160, as mentioned in its source [11].

Performance evaluation of the proposed failure recovery mechanism has been done in two phases. Initially, in order to test the effectiveness of the proposed model, we have evaluated it for its failure detection and recovery time in situations concerning both single and multi-node failures for the first three datasets, viz., Dataset 1, 2 and 3. We have compared the results on these three datasets to those obtained in a non-clustered environment, i.e., when no base clustering is used. Subsequently, for all the four datasets, the proposed method has been compared with a state-of-the-art checkpoint-based failure recovery mechanism proposed in [11].

While comparing performance against a non-clustered environment, we have observed that for all the datasets, if the nodes are not partitioned into clusters, then only single node failures can be detected and recovered from. Further, to test

for single and multi-node failures, we have randomly selected one or more line(s) in the log file of a node at which the failure would occur and recorded the time required for failure detection and recovery. For evaluation of single node failure detection, we have allowed a node to fail at the 20th and 40th lines of the log file and recorded the total time required for failure detection. As shown in Fig. 2, the proposed method has been proved to be efficient enough since for simultaneous failures at both the 20th and 40th line of the log file, the total detection time required for the node, using the proposed method, has been much lower than the total detection time required when no base clustering is used. This is true for all three datasets. We have repeated the experiments multiple times in order to ensure robustness of the proposed method. On a similar note, when evaluated for single node failure recovery, we have once again observed that for all the three datasets, total recovery time required using the proposed method, has been much lower than that required when no base clustering is used, as shown in Fig. 3.

As established in exiting works like [24], multi-node failure recovery is not possible if no base clustering is used. Hence, subsequently, we have evaluated the proposed failure recovery method for multi-node failure detection and recovery. We have observed that for all the three datasets, if two nodes fail simultaneously at the 20th line and then at the 40th line, the total time required for detection and recovery when failure occurs at the 40th line, has been much higher than that required when failure occurs at the 20th line. These results have been illustrated in Fig. 4.

Likewise, in the next step of evaluation, while comparing the performance of the proposed failure recovery mechanism against the state-of-the-art checkpoint-based failure recovery mechanism, we have allowed a node to fail at the 20th and 40th lines of the log file in order to evaluate the performance with respect to single node failure recovery. We have then recorded the total time required for node recovery. As illustrated in Fig. 5, the proposed method has been proved to be efficient enough as compared to the checkpoint-based method since for single node failures at both the 20th and 40th lines of the log file, the total recovery time required for the node using the proposed method has been significantly less than that required using the checkpoint-based method. This is true for all four datasets. The reason behind this is that the proposed method considers each user as a separate node and remains unaffected by reassignment or recomputation. The robustness of the proposed method has been ensured by repeating the experiments multiple times. Similarly, when evaluated for multi-node failure recovery, for all four datasets, the total recovery time required using the proposed method has been significantly lower than that required using the checkpoint-based algorithm, as shown in Fig. 6.

Additionally, we have also compared the storage requirement for the proposed methodology against other existing methods, viz., the failure recovery method that does not use the base clustering step [24] and the check-point based failure recovery method [11]. Table 2 shows that the storage requirement for the proposed mechanism is linear, only comparable to the non-clustered approach which however, is not capable of handling multi-node failures. The checkpoint-based recovery method, on the other hand, has higher storage requirement than the proposed method.

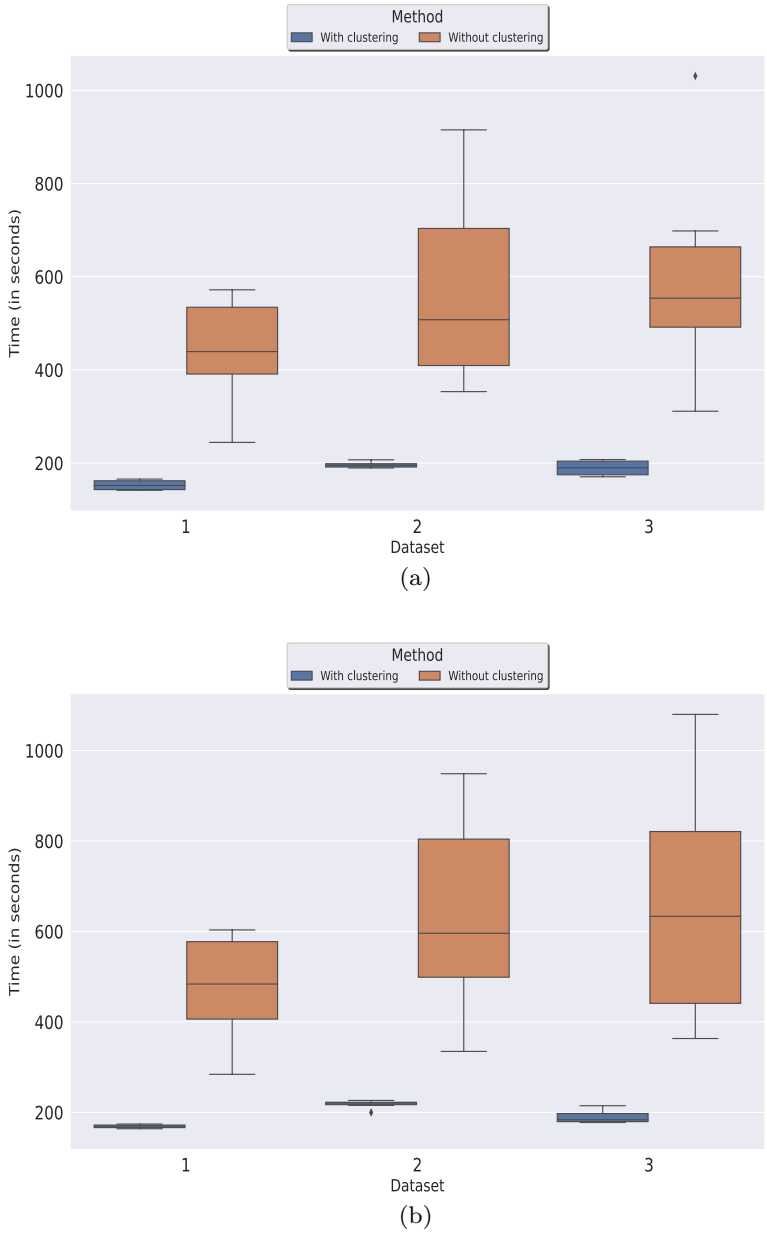
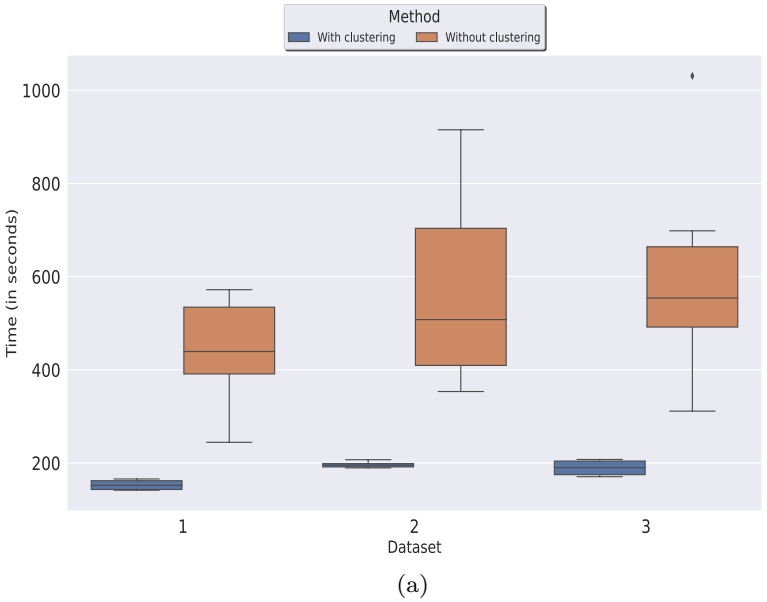
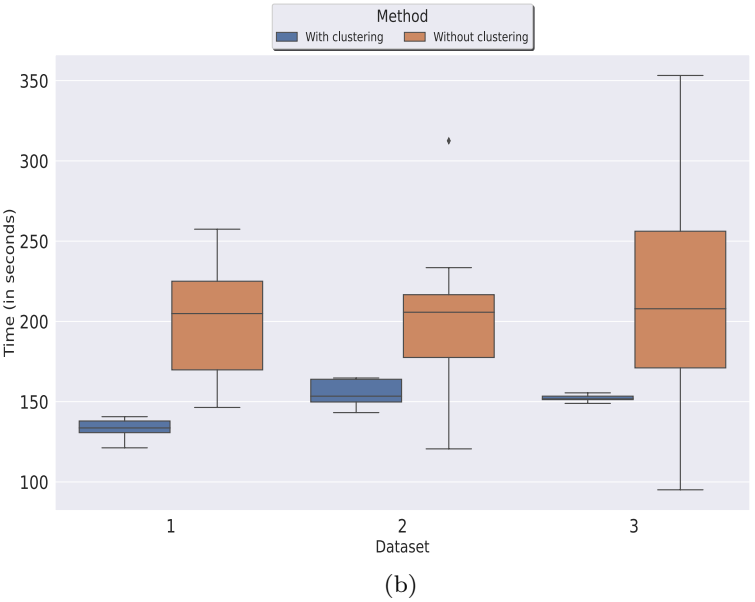


Fig. 2 **a** Shows detection time required for single node failure (at the 20th line) for all three datasets with and without clustering; **b** Shows detection time required for single node failure (at the 40th line) for all three datasets with and without clustering

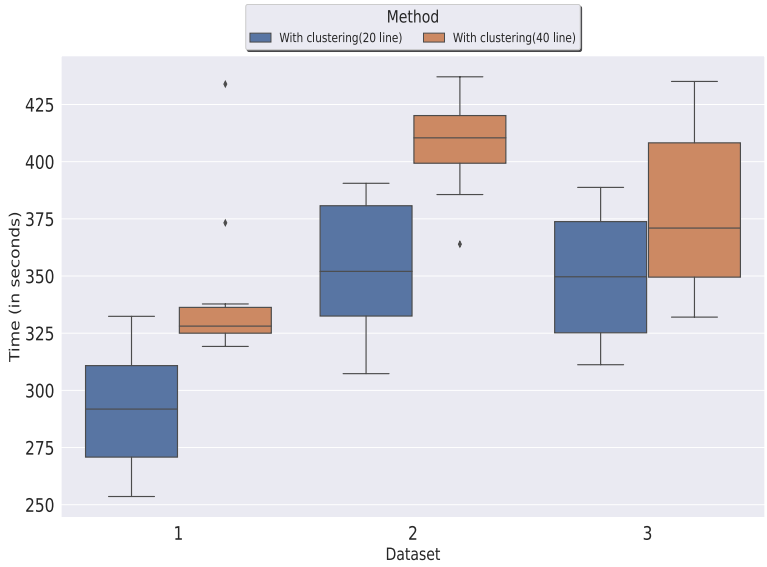


(a)

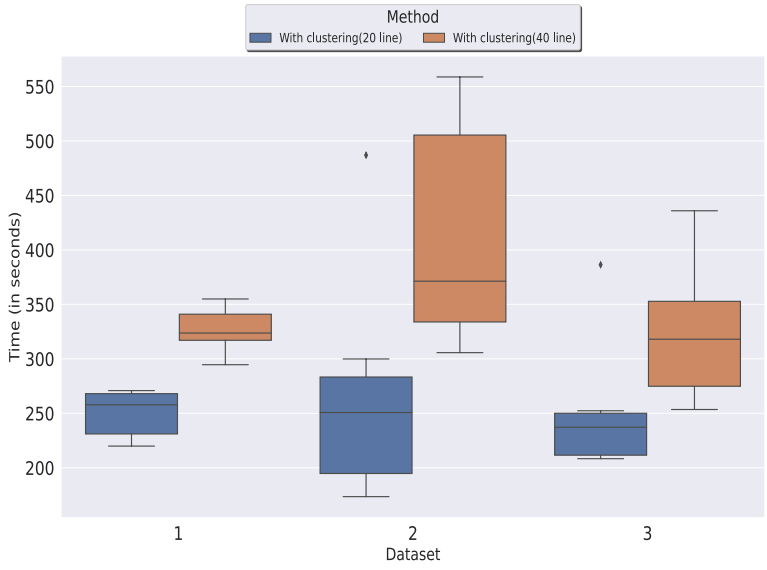


(b)

Fig. 3 **a** Shows recovery time required for single node failure (at the 20th line) for all three datasets with and without clustering; **b** shows recovery time required for single node failure (at the 40th line) for all three datasets with and without clustering



(a)



(b)

Fig. 4 **a** Shows detection time required for multi-node failure when the node fails at the 20th line as compared to when the node fails at the 40th line; **b** shows recovery time required for multi-node failure when the node fails at the 20th line as compared to when the node fails at the 40th line

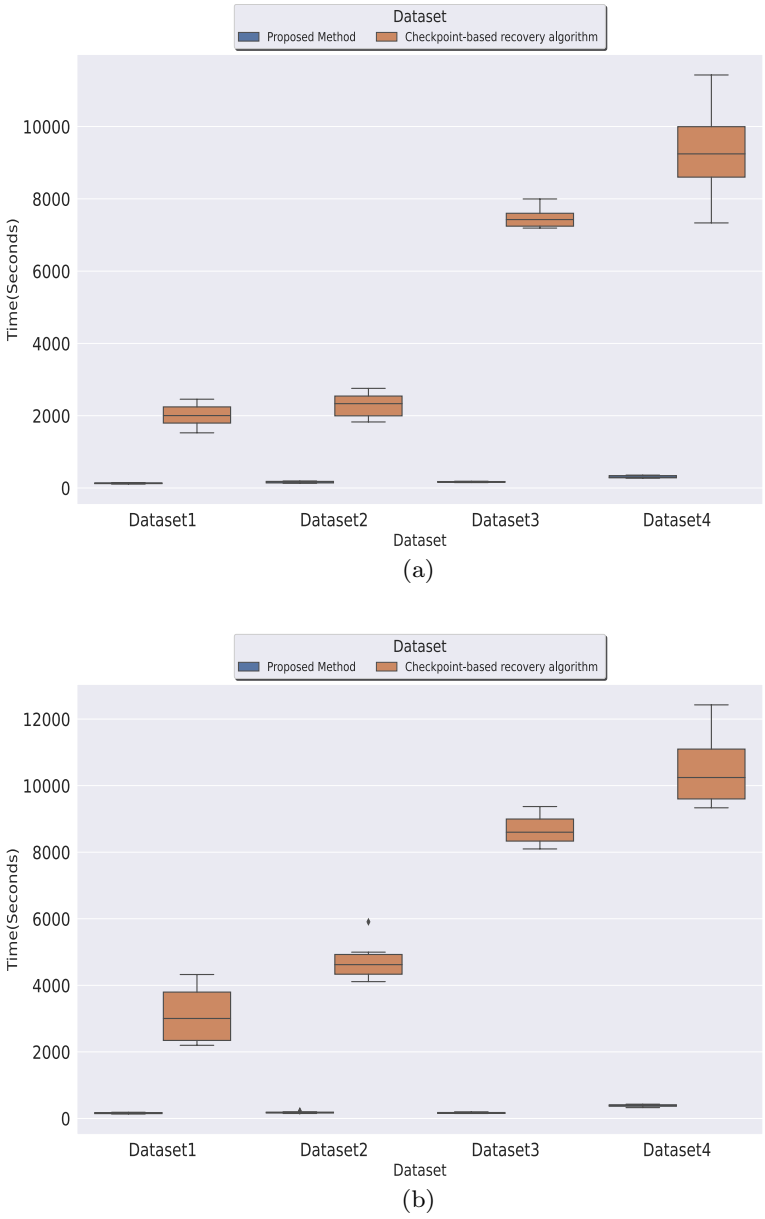
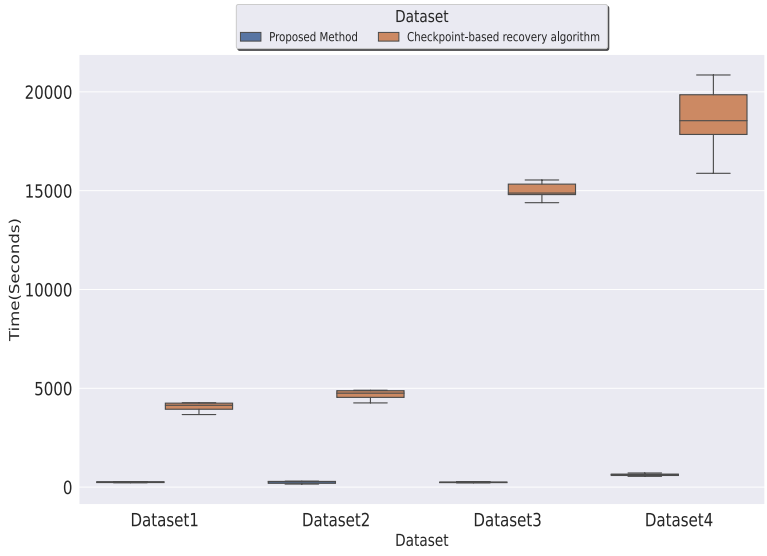
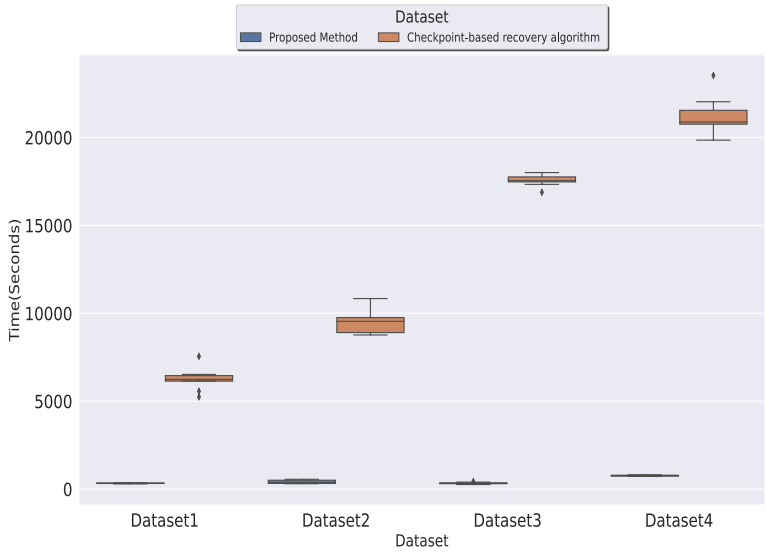


Fig. 5 **a** Shows recovery time required for single-node failure for the four datasets in comparison when the node fails at the 20th line; **b** shows recovery time required for single-node failure for the four datasets in comparison when the node fails at the 40th line, comparison performed against checkpoint-based recovery algorithm proposed in [11]



(a)



(b)

Fig. 6 **a** Shows recovery time required for multi-node failure for the four datasets in comparison when the node fails at the 20th line; **b** shows recovery time required for multi-node failure for the four datasets in comparison when the node fails at the 40th line, comparison performed against checkpoint-based recovery algorithm proposed in [11]

Table 2 Space complexity of the proposed methodology as compared to other existing methods for failure recovery

Methods	Complexity	Terminology
Proposed methodology	$O(s \times m \times k)$	m ->Number of nodes within each cluster, k -> Number of clusters, s-> Size of log file
Non-clustered approach	$O(N \times p \times s)$	N ->Total number of nodes, p-> Number of neighbors (Here, $p >> k$ and $N >> m$)
Checkpoint-based algorithm	$O(\|P\| + \|P\| \times \|N\| + \|P\|^2)$	P-> Partition (set of vertices), N -> Total number of compute nodes

9 Discussion and conclusion

In this work, we have developed a scalable, failure-recovery model for distributed graph processing systems which is capable of addressing the problem of both single and multi-node failure detection and recovery. The proposed failure recovery mechanism, when evaluated on four benchmark datasets, has consistently shown better performance as compared against existing methods that use a non-clustered environment as well as against the checkpoint-based failure recovery method.

The advantages of the proposed fault-recovery method are many-fold. Firstly, it is an improvement over existing GPSs where graphs are partitioned into several sub-graphs using edge-cut or vertex-cut, and sub-graphs residing within failed nodes are redistributed over multiple healthy nodes, which indirectly leads to increase in communication cost.

Secondly, the method proposed in the current scope of work also improves computational latency by avoiding creation and maintenance of replicas.

Thirdly, the recomputation overhead is avoided by not assigning a failed sub graph to a healthy node.

Fourthly, the log file is split up and dispersed among some virtual clusters unlike SSB where the log file is disseminated to all neighbour nodes in the network. This reduces the storage cost significantly.

Moreover, on failure, a node receives back up not from all neighbouring nodes, but only one from each neighbouring cluster. This drastically reduces the time needed for recovery of a node (see Theorem 2).

Further, the proposed model has been able to detect and recover from multi-node failures since it does not depend on neighbours from its own cluster. Thus, distributing backup files into multiple clusters ensures better availability and promises recovery of multiple nodes simultaneously (see Theorem 3).

Another important advantage of the proposed method is that it considers each user/participant as an individual node and employs an agglomerative hierarchical clustering algorithm to partition the nodes. Each virtual cluster, thus created, consists of a set of nodes or users. Communication between these nodes have been maintained using log replication which increases flexibility since it does not require

a persistent storage [16, 19]. Records of individual users are stored in single-write, append-only data structures which also reduces memory overhead.

Thus, the proposed strategy is a decentralized one that demands little participant collaboration avoiding run-time inspections or configuration management. Additionally, log replication creates a distributed system with high resilience by ensuring that every interacting component carries a durable copy of the data which is an add-on service achieved over conventional distributed systems.

The current scope of work, however has a few limitations. In the proposed method, on receiving a chunk from a node in a neighbouring cluster, the imitator node distributes the chunk to two nodes (chosen randomly) within its own cluster, in order to restore backup if one fails. However, if both these nodes fail simultaneously, then only partial recovery is possible by the proposed recovery mechanism. This is also the case when an ordinary node fails during chunk distribution.

Nevertheless, in decentralised applications, where addition of users or participants in real time is critical, the proposed model would make sure that a new node be added quickly and its empty log file be bootstrapped from other nodes without delay. It would also be beneficial to applications that use local networks between small groups of users where clients within a group communicate only over the network local to that group, for example, a team chatting application. Thus, the proposed failure-recovery model provides enhanced performance and is an improvement over conventional centralized systems that have high storage overhead and suffer from single point of failure.

Acknowledgements AM is a Senior Research Fellow supported by the Visvesvaraya Ph.D. Scheme for Electronics and IT, under Ministry of Electronics and Information Technology, Government of India. NC acknowledges the DST, ICPS project grant T-884 on “Connected Smart Health Services for Rural India”.

Author Contributions Conceptualization of Methodology: AM, RC, NC. Data Curation, Data Analysis, Formal analysis, Visualization, Investigation, Implementation, Validation, Original draft preparation: AM. Methodology Validation, Reviewing and Editing: RC, NC. Overall Supervision: RC, NC.F

Funding This work has not received any funding.

Data availability Datasets 1 and 2 can be freely downloaded from https://snap.stanford.edu/data/email-Eu-core-temporal.html. Dataset 3 is available at https://snap.stanford.edu/data/ego-Facebook.html. Dataset 4 can be downloaded from https://snap.stanford.edu/data/com-LiveJournal.html.

Code availability The proposed failure recovery mechanism has been implemented in Python3 and is freely available at https://github.com/aradhita1988/Failure-recovery.

Declarations

Conflict of interest The authors declare no competing interest.

References

1. Huang J, Qin W, Wang X, Chen W (2020) Survey of external memory large-scale graph processing on a multi-core system. J Supercomput 76(1):549–579

2. Chen R, Yao Y, Wang P, Zhang K, Wang Z, Guan H, Zang B, Chen H (2017) Replication-based fault-tolerance for large-scale graph processing. IEEE Trans Parallel Distrib Syst 29(7):1621–1635
3. Le QV (2013) Building high-level features using large scale unsupervised learning. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE, pp. 8595–8598
4. Dobre C, Xhafa F (2014) Parallel programming paradigms and frameworks in big data era. Int J Parallel Prog 42(5):710–738
5. Low Y, Gonzalez JE, Kyrola A, Bickson D, Guestrin CE, Hellerstein J (2014) Graphlab: a new framework for parallel machine learning. arXiv preprint arXiv:1408.2041
6. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135–146
7. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) Powergraph: distributed graph-parallel computation on natural graphs. In: 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), pp. 17–30
8. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2012) Distributed graphlab: a framework for machine learning in the cloud. arXiv preprint arXiv:1204.6078
9. Lu W, Shen Y, Wang T, Zhang M, Jagadish HV, Du X (2018) Fast failure recovery in vertex-centric distributed graph processing systems. IEEE Trans Knowl Data Eng 31(4):733–746
10. Zhao Y, Yoshigoe K, Xie M, Bian J, Xiong K (2020) L-powergraph: a lightweight distributed graph-parallel communication mechanism. J Supercomput 76(3):1850–1879
11. Shen Y, Chen G, Jagadish H, Lu W, Ooi BC, Tudor BM (2014) Fast failure recovery in distributed graph processing systems. Proc VLDB Endow 8(4):437–448
12. Margo D, Seltzer M (2015) A scalable distributed graph partitioner. Proc VLDB Endow 8(12):1478–1489
13. Robinson DC, Hand JA, Madsen MB, McKelvey KR (2018) The Dat Project, an open and decentralized research data tool. Scientific data 5(1):1–4
14. Blähser J, Göller T, Böhmer M (2021) Thine-approach for a fault tolerant distributed packet manager based on hypercore protocol. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, pp. 1778–1782
15. Robinson DC, Hand JA, Madsen MB, McKelvey KR (2018) The dat project, an open and decentralized research data tool. Sci Data 5:180221. https://doi.org/10.1038/sdata.2018.221
16. Tarr D, Lavoie E, Meyer A, Tschudin C (2019) Secure scuttlebutt: an identity-centric protocol for subjective and decentralized applications. In: Proceedings of the 6th ACM Conference on Information-Centric Networking, pp. 1–11
17. Tsipenyuk GY (2018) Evaluation of decentralized email architecture and social network analysis based on email attachment sharing. Tech. rep., University of Cambridge, Computer Laboratory, https://doi.org/10.17863/CAM.21035
18. Sandoval IV, Atashpendar A, Lenzini G, Ryan PY (2021) Pakemail: authentication and key management in decentralized secure email and messaging via pake. arXiv preprint arXiv:2107.06090
19. Kermarrec AM, Lavoie E, Tschudin C (2020) Gossiping with append-only logs in secure-scuttlebutt. In: Proceedings of the 1st International Workshop on Distributed Infrastructure for Common Good, pp. 19–24
20. Paul HS, Gupta A, Sharma A (2006) Finding a suitable checkpoint and recovery protocol for a distributed application. J Parallel Distrib Comput 66(5):732–749
21. Dathathri R, Gill G, Hoang L, Pingali K (2019) Phoenix: a substrate for resilient distributed graph analytics. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 615–630
22. Tschudin C (2019) A broadcast-only communication model based on replicated append-only logs. ACM SIGCOMM Comput Commun Rev 49(2):37–43
23. Singh A, Ngan TW, Druschel P, Wallach DS (2006) Eclipse attacks on overlay networks: threats and defenses. In: Proceedings IEEE INFOCOM 2006 25TH IEEE International Conference on Computer Communications, pp. 1–12
24. Roy C, Chakraborty D, Debnath S, Mukherjee A, Chaki N (2021) Single failure recovery in distributed social network. In: Hong T, Wojtkiewicz K, Chawuthai R, Sitek P (eds) Recent Challenges in Intelligent Information and Database Systems - 13th Asian Conference, ACIIDS 2021, Phuket, Thailand, April 7-10, 2021, Proceedings, Springer, Communications in Computer and Information Science, vol. 1371, pp. 203–215, https://doi.org/10.1007/978-981-16-1685-3_17

25. Peluso S, Romano P, Quaglia F (2012) Score: a scalable one-copy serializable partial replication protocol. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, pp. 456–475
26. Schiper N, Sutra P, Pedone F (2010) P-store: genuine partial replication in wide area networks. In: 2010 29th IEEE Symposium on Reliable Distributed Systems, IEEE, pp. 214–224
27. Kalavri V, Vlassov V, Haridi S (2017) High-level programming abstractions for distributed graph processing. IEEE Trans Knowl Data Eng 30(2):305–324
28. Murtagh F, Contreras P (2017) Algorithms for hierarchical clustering: an overview, ii. Wiley Interdiscipl Rev Data Mining Knowl Discov 7(6):e1219
29. Day WH, Edelsbrunner H (1984) Efficient algorithms for agglomerative hierarchical clustering methods. J Classif 1(1):7–24
30. Shahapure KR, Nicholas C (2020) Cluster quality analysis using silhouette score. In: 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA), IEEE, pp. 747–748
31. Wang X, Xu Y (2019) An improved index for clustering validation based on silhouette index and calinski-harabasz index. In: IOP Conference Series: Materials Science and Engineering, IOP Publishing, vol. 569, p. 052024
32. Paranjape A, Benson AR, Leskovec J (2017) Motifs in temporal networks. In: Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, pp. 601–610
33. Leskovec J, Mcauley J (2012) Learning to discover social circles in ego networks. In: Pereira F, Burges C, Bottou L, Weinberger K (eds) Advances in neural information processing systems, vol 25. Curran Associates Inc., Red Hook
34. Yang J, Leskovec J (2012) Defining and evaluating network communities based on ground-truth. In: Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, pp. 1–8
35. Besta M, Podstawski M, Groner L, Solomonik E, Hoefler T (2017) To push or to pull: On reducing communication and synchronization in graph computations. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp. 93–104
36. Chatterjee M, Mitra A, Setua SK, Roy S (2020) Gossip-based fault-tolerant load balancing algorithm with low communication overhead. Comput Electr Eng 81:106517