Computers in Industry 99 (2018) 156–172

Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.elsevier.com/locate/compind

Fault tolerance in cloud computing environment: A systematic survey

Moin Hasan*, Major Singh Goraya

Department of Computer Science and Engineering, Sant Longowal Institute of Engineering and Technology, India

ARTICLE INFO

Keywords:
Cloud computing
Faults and failures
Fault tolerance
Survey

ABSTRACT

Fault tolerance is among the most imperative issues in cloud to deliver reliable services. It is difficult to implement due to dynamic service infrastructure, complex configurations and various interdependencies existing in cloud. Extensive research efforts are consistently being made to implement the fault tolerance in cloud. Implementation of a fault tolerance policy in cloud not only needs specific knowledge of its application domain, but a comprehensive analysis of the background and various prevalent techniques also. Some recent surveys try to assimilate the various fault tolerance architectures and approaches proposed for cloud environment but seem to be limited on some accounts. This paper gives a systematic and comprehensive elucidation of different fault types, their causes and various fault tolerance approaches used in cloud. The paper presents a broad survey of various fault tolerance frameworks in the context of their basic approaches, fault applicability, and other key features. A comparative analysis of the surveyed frameworks is also included in the paper. For the first time, on the basis of an analysis of various fault tolerance frameworks cited in the present paper as well as included in the recently published prime surveys, a quantified view on their applicability is presented. It is observed that primarily the checkpoint-restart and replication oriented fault tolerance techniques are used to target the crash faults in cloud.

1. Introduction

Cloud computing has been prominently existing as an on-demand computing service paradigm and immensely benefiting the small-scale users as well as large-scale commercial and scientific applications. It is defined as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. On-demand access, resource autonomy, rapid elasticity and always-on availability are the primary characteristics of cloud computing [2]. Cloud resources are provisioned using standard protocols (IAM, OAuth, OpenID, etc. for authentication; and AMI, OVF, SOAP, REST, etc. for data and workload migration [3]) to create the wider acceptability of cloud services. Besides this, cloud offers greater business agility at the reduced cost which further attracts a vast user base. A recent survey conducted over 433 enterprise respondents containing 1000+ employees reveals that 95% of the respondents are using cloud [4]. Kazarian et al. [5] reported 91% adoption of cloud by the IT professionals in more than 3000 small and midsize businesses. Anticipating its vast benefits, distinguished IT organizations (such as Amazon, Microsoft, IBM, Google, Yahoo, etc.) are into the foray to deliver cloud services.

Though, cloud has gathered much attention over the time, but it is still considered adolescent in terms of fault handling capability [6]. The cloud computing architecture is dynamic and growing in complexity [7–9]. Its deployment uses millions of commodity components rather than conventional ones [10]. Due to this, it is always prone to faults and failures. Fault is an abnormal condition or defect in one or many parts of a system, which may result in the inability of the system to perform its intended functions [11]. Fault occurrence creates error in the system. Error is defined as a deterioration in one or more system components and creates difference between normal and actual state of the system [12]. The errors lead the system to failure, which interrupts the normal delivery of the services and degrades the system performance. Improper handling of system failures may lead the system to an unworkable state [11]. The effects are so adverse at times that they could traumatize the economic state of the service provider. For instance, in 2013, a breakdown of just about 45 min resulted in an economic loss of $5 million to Amazon cloud [13]. It may be one of the reasons for the reluctance of a big pool of users towards acquiring cloud services and makes fault tolerance as one of the most imperative issues in cloud computing.

Fault tolerance is defined as the capability of a system to keep performing its intended task even in the presence of faults [14,15]. Without fault tolerance capability, even a well-designed system with best of the

* Corresponding author.
E-mail addresses: mmoinhhasan@gmail.com (M. Hasan), mjrsingh@yahoo.com (M.S. Goraya).

https://doi.org/10.1016/j.compind.2018.03.027
Received 14 July 2017; Received in revised form 16 January 2018; Accepted 20 March 2018

components and services cannot be considered as reliable [16]. Reliability is a highly significant facet of cloud, as a large number of delay sensitive (real-time) applications are to be executed. Moreover, service reliability is imperative to the wider acceptability of cloud. Therefore, the issue of fault tolerance has got a considerable attention in research and numerous fault tolerance frameworks have been proposed in literature over the period. Through this paper, we endeavour to present a survey of fault tolerance in the cloud computing environment.

1.1. Motivation of the survey

In the literature, we observe that despite extensive research in the field of fault tolerance in cloud, only a few surveys [17–19,16,20,21] have been published. Although, these surveys have considerable contribution in the field, but in themselves, do not seem to be exhaustive and comprehensive. These surveys appear to be limited in respect of one or other account.

Cheraghlou et al. [16] gave only a brief description of different fault tolerance techniques without focusing on fault types. Further, the discussion of only a few number of frameworks in the survey limits its scope. [21] focused directly on fault tolerance and classified fault tolerance policies as *exclusively handled* and *collaboratively handled*. This survey does not provide view on the conventional classification of the fault tolerance models in cloud. Agarwal and Sharma [17] gave the taxonomies of fault, error and failure; but missed the theoretical explanation. The authors have not included any existing fault tolerance framework in the survey to strengthen the discussion of fault tolerance techniques. Ataallah et al. [19] included a brief description of various fault tolerance parameters in their survey, but failed to include the description of fault types. Very limited frameworks have been explained in this survey which are insufficient to describe the state-of-the-art. Several types of faults and fault tolerance techniques are briefly described in the survey given by Saikia and Devi [20]. However, authors have not given any classification of the described faults and fault tolerance techniques in the survey. Further, only a few fault tolerance frameworks are included in the survey without citing any comparative analysis. Amin et al. [18] also enlisted various fault tolerance metrics along with a brief description about fault detection. Again, very limited fault tolerance frameworks are explained without any reflection of the methodology of fault tolerance used in the frameworks.

It can apparently be concluded that none of the above cited surveys presents the complete structure of fault tolerance in cloud computing. In order to understand the complete structure of fault tolerance the readers have to refer different sources. Therefore, we motivated to write a comprehensive and systematic survey on fault tolerance in cloud by describing its complete structure which includes the description of (a) various fault types, their causes and classification; (b) fault tolerance approaches and techniques; and (c) fault tolerance frameworks. Table 1. summarises and present a comparative analysis of the existing surveys cited in this paper and the present survey in the context of inclusion (✓) and non-inclusion (×) of the attributes: fault taxonomy, fault tolerance approaches, fault tolerance frameworks, comparative analysis, and graphical representation.

1.2. Scope of the survey

Scope of the present survey is:

- Description of various fault types and their causes in cloud computing environment.
- Description of basic fault tolerance approaches used in cloud computing environment.
- Description of different fault tolerance frameworks proposed in literature for cloud computing environment.

Fault types are explained in a tabular form for the ease of understanding. A comparative analysis of the surveyed frameworks is also given which focuses on the basic approach, methodologies used, fault applicability and key features.

1.3. Survey plan and organization

The survey plan broadly includes article selection, fault classification, identification of fault tolerance approaches and methods, description of fault tolerance frameworks, discussion and future directions. The survey plan is executed through multiple phases described as follows:

- *Phase-1 (Articles Selection)*: In the first phase number of research articles (including surveys) related to the field are collected from reputed sources. The collected articles are carefully examined and filtered based on their *titles*, *abstracts*, and *research contributions*. While examining the research contributions, the novelty and quality of the work is critically analysed. The articles (regarding each fault tolerance method) for inclusion in the paper are selected with the criteria that the reader would be able to know the basic implementations and possible modifications/customization of each fault tolerance method. Necessary efforts are made to assure and maintain the diversity of the articles in order to remove the ambiguity and enhance the knowledge base of the readers.
- *Phase 2 (Fault Classifications in Cloud)*: In the second phase, the collected articles are intensely scrutinized to identify different fault types in cloud. The identified fault types are thoroughly analysed for their categorization. Section 2 includes the brief description of different fault types, their root causes, and classification in cloud.
- *Phase 3 (Fault Tolerance Approaches in Cloud)*: In this phase, the collected articles are further analysed to identify various fault tolerance approaches in cloud. The identified approaches are enlisted and described in Section 3 of the survey. The fault tolerance methods based on the identified approaches are also explained and hierarchically presented in Section 3.
- *Phase 4 (Fault Tolerance Frameworks in Cloud)*: This phase contains the core research contribution of this survey to explain various fault tolerance frameworks proposed in the literature. The objective is to provide an evolutionary knowledge base in such a way that the research contribution towards each fault tolerance method could be covered. Section 4 explains various prominent fault tolerance

Table 1
Comparison of the present survey with the cited surveys.

Survey Paper	Fault Taxonomy	Fault Tolerance Approaches	Fault Tolerance Frameworks	Comparative Analysis of Frameworks	Graphical Representation of Results
Cheraghlou et al [16]	×	✓	✓	✓	×
Tchana et al. [21]	×	✓	✓	×	✓
Agarwal and Sharma [17]	✓	✓	×	×	×
Ataallah et al.[19]	×	✓	✓	✓	×
Saikia and Devi [20]	✓	×	✓	×	×
Amin et al. [18]	×	✓	✓	×	×
Present Survey	✓	✓	✓	✓	✓

frameworks in the context of basic implementation details, fault applicability, evaluation methodology, and key features.

- *Phase 5 (Survey Discussion, Future Directions, and Conclusions)*: In the last phase, surveyed fault tolerance frameworks are deeply analysed to present the survey results. This phase also includes research directions and survey conclusions. Section 5 discusses various observations and results drawn from the survey, which includes:
  - a quantified view on how much a particular fault category is targeted in the research.
  - a quantified view on how much a particular fault tolerance method is utilized in the research.
  - a comparative analysis of various fault tolerance frameworks described in the survey.

Based on the existing challenges observed from the literature, probable future research directions concerning fault tolerance in cloud are given in Section 6. Finally, the paper is concluded in Section 7.

2. Fault classification and architecture of fault tolerance in cloud

2.1. Fault classification in cloud

Jhawar and Piuri [10] classified the faults into two broad categories as *crash faults* and *byzantine faults*, described as follows

- Crash Faults: The faults occur due to the failure of one or more system components (e.g., power supply, storage disks, memory chips, processors, network switches and routers, etc.) are called crash faults. Occurrence of such faults affects the physical structure of the system and manual interaction is generally required for the repair of the affected component(s). Software agents may be deployed to tolerate these faults.
- Byzantine faults: The faults which create an ambiguity in the expected output resulting in unpredictable conditions are known as byzantine faults. Occurrence of byzantine faults affects the logical structure of the system and does not require manual interaction as they can be handled through proper fault tolerance strategies. It may be quite difficult to properly examine and conclude the byzantine conditions due to the complexity and diversity existing in the system.

Cloud possesses a distributed architecture, therefore, its faults are almost similar to those in other distributed computing paradigms [10]. Numerous types of faults are pointed out in the literature specific to the cloud environment [22,12,13]. These faults can be classified into the above two categories. However, certain fault types may be categorized as both crash as well as byzantine faults as shown in Fig. 1. Table 2 presents the brief description of various fault types in cloud.

2.2. Architecture of fault tolerance in cloud

The cloud architecture is broadly comprised of a physical layer and an abstraction layer [23,1]. The Physical layer includes hardware components (CPU, storage, network, etc.) upon which the abstraction layer is deployed to provide the cloud services. Corresponding to the delivered services the abstraction layer is further divided into Infrastructure as a Service (IaaS) layer, Platform as a Service (PaaS) layer, and Software as a Service (SaaS) layer. [1]. The failure of any hardware component in the physical layer affects the abstraction layer and consequently the service delivery. Due to the interdependencies in the service delivery layers, fault tolerance in cloud follows layered architecture where fault in one layer may affect its upper layer(s) [24] as shown in Fig. 2. Faults in the PaaS layer may affect the SaaS layer and faults in IaaS layer may affect both PaaS and SaaS layers. For example, if the operating system installed at the platform crashes, the applications running on the operating system at the SaaS layer will also stop working. Similarly, if the hard drive of a host at the IaaS layer burns out, the operating system installed on the host at the PaaS layer will also crash, which in turn will stop its running applications at the SaaS layer.

Faults erupted in the physical layer may have the severest consequences and a robust fault tolerance is required at the infrastructure level for the seamless service provisioning. However, certain faults are specific to their respective layers, independent of the underlying architecture. For example, a run-time exception at PaaS or a virus attack at SaaS. The general fault tolerance procedure in cloud environment is as follows:

The fault detection mechanism is applied at the respective service layers to detect their probable faults. Corresponding to a services layer (IaaS/PaaS/SaaS), fault types, or application environment, different fault detection mechanisms have been adopted in the research [16]. For example, faults at IaaS are generally detected through health monitoring techniques, like heartbeat protocol [25]. Many of the byzantine faults are detected through voting mechanisms. Various fault detection mechanisms used in the literature along with the fault tolerance frameworks are discussed in Section 4. After fault detection at a specific layer, its corresponding fault tolerance procedure is called to neutralize/minimize its effect on service. The successful service delivery is achieved if no fault occurs on any layer. A non-detected fault may result in failure of service. In such case, the other measures may be adopted to minimize its effect on the service. If required service rescheduling [26] may also be applied.

3. Fault tolerance approaches in cloud

The fault tolerance approaches are broadly classified into two basic

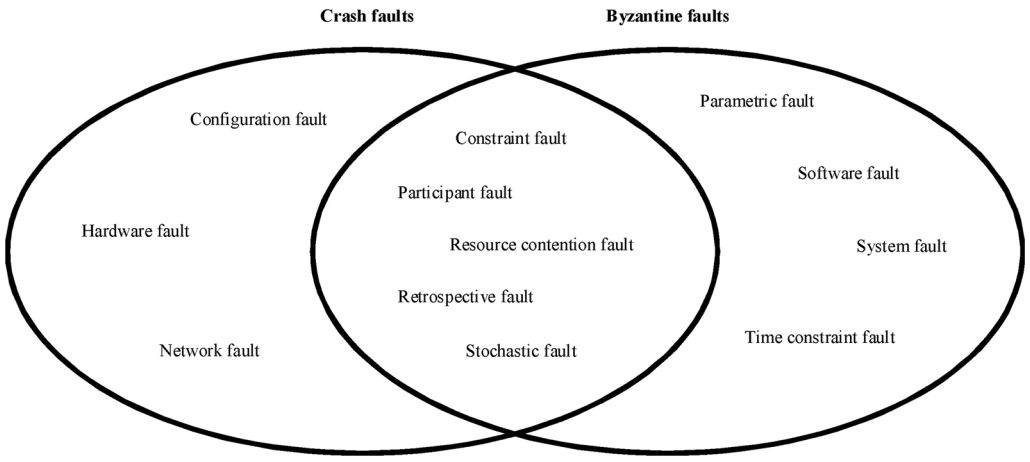

Fig. 1. Fault categorization in cloud.

Table 2
Categorization of various fault types in cloud.

Fault Type	Category (Crash/Byzantine)	Brief Description
Configuration fault	Crash	Occurs when the ordering of the system components is disturbed
Constraint fault	Both	Occurs when a fault condition arises and ignored by the responsible agent
Hardware fault	Crash	Occurs at the infrastructure level in the service delivery model of the cloud system due to the failure of any hardware component
Network fault	Crash	Occurs due to the failure in either of network components (e.g., switches, routers, etc.)
Parametric fault	Byzantine	Occurs due to unknown variation in the parameters
Participant fault	Both	Occurs due to the conflict between cloud participants like consumer, provider, administrator, etc.
Resource contention fault	Both	Resultant of the conflict when a resource is being shared for the access
Retrospective fault	Both	Occurs due to the lack of information about the past behavior of the system
Software fault	Byzantine	Generally, the resultants of software updates in the system
Stochastic fault	Both	Occurs due to insufficient statistical information to assess the system state
System fault	Byzantine	Occurs due to incomplete knowledge of the processes that control the service provisioning in the system
Time constraint fault	Byzantine	Refers to the situation which causes an application unable to complete before the specified deadline

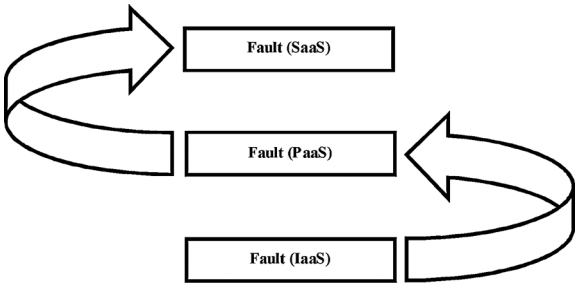

Fig. 2. Fault relationship in cloud.

categories, viz. *proactive* and *reactive* [17,18,27,16,28,29]. Based on these approaches, several methodologies have been adopted in the literature to achieve fault tolerance. They are discussed as follows and shown hierarchically in Fig. 3.

3.1. Proactive approaches

The word *proactive* in context of fault tolerance is defined as the ability of the system to be in a prepared or controlled state for handling the possible interruptions (faults, errors, failure) before they occur. The system state in proactive approaches is continuously monitored and the fault occurrence is estimated using the artificial intelligence techniques. Necessary actions are then taken to prevent the fault occurrence. Working of these approaches is based on the experience and expectation [30]. The proactive fault tolerant systems remain uninterrupted until the expectation matches with the experience. Proactive fault tolerance in cloud environment can be achieved in the following ways [18,27,16,29]:

3.1.1. Self-Healing

It is defined as the capability of a system to have an autonomous recovery from faults by periodically applying specific fault recovery procedures consisting of supervision tasks [31]. A system can perceive the erroneous conditions independent of human interactions and can make guided adjustments to restore the normal state [32]. It is inspired from the biological fact, “How organisms manage to survive in difficult situations [33,34].” A self-healing fault tolerant system requires various fault aspects (location, duration, intensity, etc.) to be looked carefully for its successful action. The performance of self-healing systems improves with time as different faults are encountered which enhances the system experience.

3.1.2. Pre-emptive migration

It is defined as the capability of a system to proactively move the computation away from suspicious computing nodes [35,36]. In this method, pre-fault indicators over the system are used to predict the occurrence of any fault in the nodes in near future. The tasks from fault probable nodes are pre-emptively shifted to other nodes [30]. Fault tolerant systems based on pre-emptive migration follow more probabilistic approach and usually focus on the failure rates of invidual computing nodes rather than focusing on the various fault aspects.

3.1.3. System rejuvenation

It is a process of taking periodic backup of the system. After each backup, the system is cleaned and repaired from any kind of bugs and errors. The backup is then reinstated, and a refreshed system state is achieved. There are two basic types of system rejuvenation, viz. *fixed time rejuvenation* where the time interval between any two consecutive rejuvenation events remains same and *variable time rejuvenation* where the time interval between any two consecutive rejuvenation events may vary according to the working conditions [37]. Rejuvenation can also

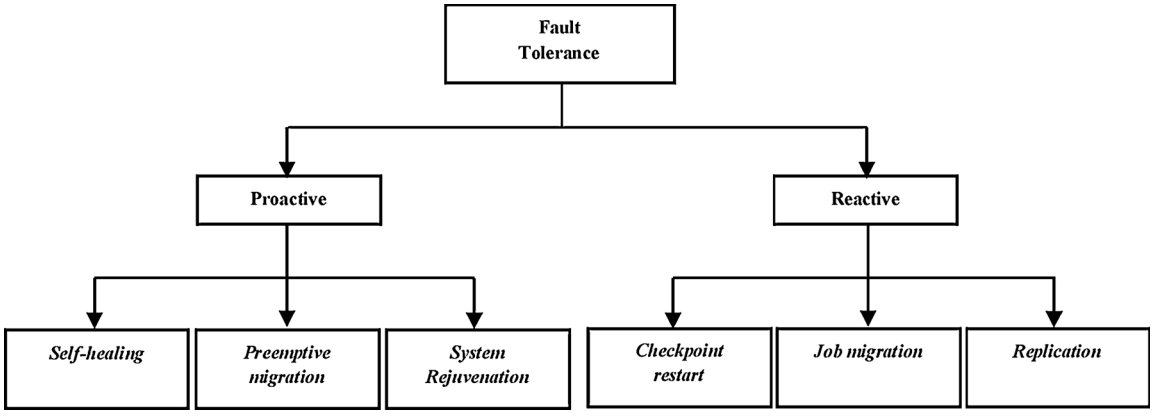

Fig. 3. Classification of fault tolerance approaches in cloud.

be categorized as *full* (all the system components are rejuvenated at a time) and *partial* (some of the system components are rejuvenated at a time). Since, cloud is a heterogeneous environment where different components may have different failure rates, partial rejuvenation at different instances of time looks to be of more use rather than full rejuvenation.

3.2. Reactive approaches

The reactive fault tolerance approaches handle the faults after their occurrence. The influences of occurred faults are abridged using system maintenance programs. The working of reactive approaches is response based rather than anticipation [30]. Reactive approaches are usually conservative in nature and need not to examine the system behaviour. Hence, they do not pose any unnecessary overhead. Reactive fault tolerance in cloud can be achieved using following methods [18,27,16,29]:

3.2.1. Checkpoint restart

Task execution states are periodically saved in this method. In case of any failure, the task is restarted from the last saved state rather than restarting from the beginning. The desired features of any checkpoint restart technique are *scalability*, *transparency*, *and portability* [38]. Checkpoint restart is one of the most popular fault tolerance method and has found great applicability in the existing fault tolerant systems due to its dual applicability. Dual applicability means that it can be used as both *stand-alone* as well as *auxiliary* fault tolerance method. The frequency of taking checkpoints can be adjusted according to the failure rates of the system components so as to optimize the overhead. Various implementations of checkpoint restart may be found through [39,40].

3.2.2. Job migration

In this method, if the resource (executing the task) fails, the task is migrated to some other similar suitable resource instance. It is generally used to tolerate crash faults. However, unlike *pre-emptive migration* (where migration takes place after the *prediction* of fault), this method is reactive as the migration takes place after the fault occurrence. Job migration usually associates communication overheads only and traditionally used for soft deadline applications. *Retry* is a variation of job migration method, which is used in a few fault tolerance frameworks in cloud [41,26]. In retry, a failed task is executed from the beginning either at the same resource or at some other resource. In case of byzantine faults, retry can be carried out at the same resource for the sake of resource utilization. However, the number of retry attempts at the same resource can be confined according to the situation. For example, after five unsuccessful retry attempts on the same resource, the task may be migrated to some other resource.

3.2.3. Replication

In this method, the task is operated on multiple execution instances. If one instance fails, task execution remains continuous on other instance(s). It can be used to tolerate both crash as well as byzantine faults and has been used most widely. Two types of replication have been observed in literature, viz. *active replication* (both primary and backup resource(s) perform the operation) and *passive replication* (only the primary resource performs the operation and expected to produce the result, while the backup resource(s) only receive the execution updates so as to take over the operation in case the primary fails) [17,42].

Some popular variations of replication method are *recovery block*, *N-version programming*, and *active* [41]. Among all replicas for a particular application in recovery block, one is assigned as primary and others are assigned as standby. If the application on primary fails, it is sequentially restarted at the available standby replicas. Instead of restarting the application, it can be check-pointed to mitigate the waiting time. In N-version programming, the application is simultaneously executed on all replicas and the final result is decided by voting. However, different implementation may use different voting methods to obtain respective results. Active is almost similar to N-version programming except the fact that it does not use voting mechanisms to obtain the result. The final result in active is the first correctly received response.

4. Fault tolerance frameworks

This section surveys various fault tolerance frameworks proposed in the literature. Each framework is explained in adequate detail with respect to basic fault tolerance approach, methodology used, and various implementation aspects. Section 4.1 surveys renowned proactive fault tolerance frameworks, while reactive fault tolerance frameworks are surveyed in section 4.2. The abbreviations used in the survey are not standard and are decided by the authors similar to the titles of the respective frameworks.

4.1. Proactive fault tolerance frameworks

4.1.1. SHelp

Chen et al. [43] proposed a proactive fault tolerance framework using self-healing method. The authors improved the earlier proposed framework, named as ASSURE [44]. ASSURE introduced the concept of *rescue points* which are the locations in the application code in order to handle a given set of programmer-anticipated faults. These rescue points can be reused in order to handle the unanticipated faults. In this section, the ASSURE framework will be discussed first followed by the improvements made in SHelp. The ASSURE framework maintains an execution log by periodically taking the application check-points. For fault detection, it uses light-weight instrumentation mechanisms for system monitoring. For fault analysis, it uses a triage system, where the shadow of the application is deployed. Whenever a fault is detected, the application is transferred to the triage system from its latest check-pointed state and the occurred fault is reproduced there in order to recognize a suitable rescue point. The suitability of a rescue point is evaluated in terms of its *survivability* (if the selected point could rescue the application from the recurrence of fault), *correctness* (if rescue point does not introduce semantic errors), and *performance* (if rescue point does not impose significant overhead). After the verification of the rescue point, ASSURE generates a remediation patch and dynamically embed it into the software. Whenever the same fault reoccurs, the patch instantiates an appropriate rescue point and the application is rolled back to do the recovery actions.

The SHelp architecture differs from ASSURE with respect to the way of selection of the rescue points. ASSURE uses a rescue-trace graph which is traversed so as to search an appropriate rescue point. It contributes to the overall overhead of the fault tolerance. On the other hand, SHelp assigns a weight to each rescue point. Initially the weight of each point is set to zero, but it increases proportionally for a particular rescue point with respect to the number of times it is applied. Now, whenever a fault is occurred, the rescue points are searched in the decreasing order of their weights.

For performance evaluation, a prototype of SHelp is implemented on Linux and tested over various versions of four different web server applications, viz. *Apache*, *Light-HTTPd*, *ATP-HTTPd*, and *Null-HTTPd*.

Key Features: The SHelp framework offers faster functioning and comparatively lesser overhead than ASSURE. It is because of the introduction of weight assignment to the rescue points, which considerably decreases the searching time.

Limitations: The SHelp framework is limited to tolerate software faults only.

4.1.2. PFHC

Egwutuoha et al. [45] developed a proactive fault tolerance system for high performance computing (HPC) applications in cloud. It has three major modules discussed as follows:

- Node Monitoring Module: It is installed to keep an eye on each node in the cloud and is equipped with *Lm-Sensors* [46,47] which are used to develop *FTDaemon* in order to monitor various parameters like CPU temperature, fan speed, etc. Moreover, monitoring is done periodically rather than continuously so as to minimize the monitoring overhead. Whenever the parameters exceed their set value, an alarm is triggered prompting the probability of failure.
- Fault Tolerance Module: This module takes necessary fault tolerance actions if a failure is prompted. When an unhealthy node is predicted, the *FTDaemon* asks the resource provider to lease another node. The execution is then migrated to the newly added node and the administrator is informed to relinquish the unhealthy node.
- Controller Module: It is installed on every node and is responsible for the following tasks:
- *Implementation of fault tolerance policy*
- *Provides user credentials to the service provider* (c) *Live migration of VMs.*

The authors further compared the proposed system with PFHX [48] in terms of total operational cost for fault tolerance. In [48], *n* nodes are provisioned for the computation and *m* nodes are provisioned in spare (ahead of any failure prediction). Total operational cost will be the sum of operational cost of *n* computation nodes and that of *m* spare nodes, regardless whether the operation is failure-prone or failure-free. However, in PFHC, the spare nodes are only demanded whenever a fault is predicted, resulting in reduced operational cost. The performance of the proposed algorithm is evaluated in terms of execution time of different-sized HPC problems on four servers leased from *baremetalcloud* [49,50].

Key Features: Being a proactive framework, PFHC tolerates hardware faults. It is applicable to HPC applications at lower fault tolerance cost.

Limitations: It requires complex implementation to predict unhealthy nodes and then live migrate the VMs.

4.1.3. WSRC

Bruneo et al. [37] proposed a fault tolerance mechanism for Virtual Machine Manager (VMM) software in cloud using *variable-time* software rejuvenation technique. The failure model of WSRC consists of a *failure detector*, which periodically (rather than continuously) inspects the VMM for any memory mismanagement and response time fluctuation. In case of any failure, the status of all the running VMs is saved and the rejuvenation is carried out for VMM repair. The authors varied the rejuvenation time interval corresponding to the fluctuation in the workload as follows: Let T^K and δ_i are the *timer firing time* and *time interval* (when workload = i) respectively. Timer firing time is the time set for the next rejuvenation event. Now, suppose workload changes from *i to j*, which consequently changes the interval from δ_i *to* δ_j. The new firing time is then adjusted as follows:

$$T^{k+1} = t_x + \left(1 - \frac{t_x}{T^k}\right) * \delta_j$$

Where, t_x is the current time. This variable-time rejuvenation policy is then modelled using *Kronecker algebra* [51]. The VMM software operation is represented by five states, viz. *working* state, *non-detected failure* state, *detected failure* state, and *rejuvenated* state. After the completion of rejuvenation event, all the VMs are reactivated. The performance is evaluated in terms of *system availability* through simulation experiment using WebSPN tool [52] over a case study taken from earlier literature.

Key Features: The WSRC framework offers high resource availability. Rejuvenation based fault tolerance generally result in high overhead, and in case of cloud environments (with practically millions of physical hosts), it is an important concern. WSRC optimizes the fault tolerances overhead by applying variable time rejuvenation.

Limitations: WSRC can only be applied at infrastructure layer and is confined to tolerate software faults.

4.1.4. SRFSC

Liu et al. [53] proposed a proactive fault tolerance scheme for cloud applications using software rejuvenation technique. Each cloud application is considered as a combination of interconnected cloud service components, which may either be *tightly-coupled* (existing in same VM) or be *loosely-coupled* (existing in different VM). These components may communicate over a high-speed LAN via *remote procedure call*. The proposed scheme works in three steps: (a) *Aging failure detection,* (b) *Aging degree evaluation, and* (c) *Rejuvenation*. The failure is detected in terms of CPU and memory usage of certain VMs and the transmission delay. A software agent periodically collects the CPU and memory usage, encapsulate them into a packet, and transmit it to the *aging failure detector*. The expected arrival time of every next packet is then calculated by averaging the transmission time of previous *n* packets. After that the degree of failure is evaluated in terms of two aspects: (a) arrival time of next packet with respect to the expected arrival time (before expected, after expected, or lost) and (b) usage of CPU and memory. Using this information, the severity of the service component condition is divided into four levels: *Level 1*- specifies that rejuvenation is needed immediately; *Level 2*- specifies that rejuvenation should be done as soon as possible; *Level 3*- specifies that the service component should be monitored and rejuvenation is suggested; and *Level 4*- specifies that service component is performing well and no rejuvenation is needed.

If rejuvenation has to be done, the current running state of the service component is transmitted to an interim node (lies in the same network). The VM hosting the service component is also migrated to the interim node. The original VM is then rebooted and a fresh VM is now ready for the service component.

Key Features: Along with high availability, SRFSC is capable to rejuvenate multiple independent VMs.

Limitations: The failure aspects considered in SRFSC are limited to software aging and crashes.

4.1.5. FTDG

Sun et al. [54] proposed a fault tolerant scheduling framework using pre-emptive migration for stream computing. The framework architecture consists of four working spaces; viz. *user space*, *graph space*, *storm space*, and *hardware space*. Users submit their respective data streams through the user space which are transformed into DAGs (Direct Acyclic Graphs) in the graph space. In the graph space, the critical and non-critical paths of the DAGs are also determined. Scheduling mechanism including fault tolerance strategy is applied in the storm space. The hardware space consists of various datacentre resources. FTDG is proactive as it consistently monitors the arrival rate of the data streams. When there occurs a considerably large fluctuation in the arrival rate, it may affect the response time. In such case, a vertex from the critical path is selected and pre-emptively migrated to some other computing node in a way that the minimum response time would be maintained. The failure model of FTDG is comprised of failure density of both the computing nodes as well communication links.

The proposed framework is implemented on *Storm* which is an open source distributed computing platform and evaluated in terms of *reliability*, *response time*, and *throughput*.

Key Features: Response time is an extremely important aspect in stream computing and FTDG possesses the feature to maintain minimum response time.

Limitations: FTDG offers limited fault applicability (parametric faults only).

4.2. Reactive fault tolerance frameworks

4.2.1. AFTRC

Malik and Huet [55] presented a fault tolerant real-time tasks' execution model in cloud computing, named as *Adaptive Fault Tolerance in Real-time Cloud Computing* (AFTRC). In the presented model, the real-

time incoming tasks are maintained in an input buffer. Tasks in FCFS manner are then promoted for execution. Each task is replicated on *M* virtual machines, which are embedded with different algorithms for real-time task execution. The result produced by each algorithm is moved further for the acceptance test (AT), where the correctness of the result is verified. The results are then moved to the time checker (TC) so as to check whether the result is obtained before deadline or not. If none of the results is obtained before deadline, the task is sent back to input buffer. On the basis of the obtained results, the reliabilities of the corresponding virtual machines are adjusted by the reliability assessor (RA). Finally, decision mechanism module (DM) selects the output from the highest reliable node. Along with replication, checkpoint mechanism is also used in AFTRC and the task states are periodically saved in the recovery cache (RC). To evaluate the AFTRC framework, an experiment is conducted using ProActive grid interface to Amazon EC2 cloud by creating three VMs. The results are obtained by executing a series of real-time tasks for each VM in terms of their reliabilities.

Mohammed et al. [56] presented a similar fault tolerance strategy (IVFS) using replication and checkpoint-restart. It is similar in the sense that the task execution passes through the similar stages as in AFTRC. The difference is the addition of a cloud load balancer (CLB) and the way checkpoint is carried out. The CLB contains the load information regarding each VM in the cloud. After the reception of a task, the cloud controller (CC) passes it to the CLB, which replicates it on suitable VMs as per the load information. The checkpoint-restart is implemented using *Reward Renewal Process* (RRP) [40]. In RRP the overhead, recovery time, and rollback time of the checkpoint depend upon a random variable which is a function of failure rate. IVFS framework is implemented using CloudSim simulator [57].

Key Features: AFTRC promises result accuracy and is applicable for real-time applications.

Limitations: AFTRC may lead to low resource availability when the workload increases.

4.2.2. BlobCR

Nicolae and Cappello [58] proposed a two-stage checkpoint-restart mechanism for tightly coupled HPC applications in IaaS clouds using virtual disk snapshots. Instead of directly storing the checkpoints on the cloud repository, the local disks of the computing nodes are aggregated and utilized in a distributed manner, i.e., VM disk snapshots are split into equal sized *chunks* and distributed evenly over this local repository. The first working stage of the check-pointing is about saving the application state to the VM disk followed by the second stage in which the VM instance is suspended and an *optimized snapshot* of VM disk is saved to the local disk repository. To optimize the VM disk snapshots, the concepts of *shadowing* and *cloning* [59] are used. Shadowing is basically the act of storing only the differences between the last check-pointed state and current check-pointed state, instead of storing the whole object. Cloning is on the other hand is the act of duplicating an object containing all the content of the original. The first snapshot of each VM instance is cloned and the subsequent incremental differences are then written to the respective clones and then shadowed as new snapshots. For recovery purpose, application restart is optimized using *lazy transfer* [58] by fetching only the check-pointed files and the files which can be directly accessed by the guest OS as well the application.

This work is further improved in [38] by including the *live incremental snapshotting*. For this purpose, the authors utilized the concept of *copy-on-write* [60] and introduced *selective copy-on-write* principle. In *copy-on-write* technique, the chunk which has to be modified is copied to some alternate location and modifications are done on the copy instead of the original one. However, it may be disadvantageous as copying the chunks to alternate locations may result in the local repository fragmentation and degrades the performance. Moreover, it cannot be applied when chunks are required to be contiguous locations. *Selective copy-on-write* eliminates the problem of fragmentation by leveraging the fact, "Snapshotting process does not access all chunks simultaneously." Thus, if a chunk is scheduled to be accessed in future, it is copied to an alternate location, and the snapshotting process can be redirected there. However, if a chunk is actively being accessed by the snapshotting process, even *selective copy-on-write* is inapplicable and it becomes necessary to wait. BlobCR is evaluated in terms of *average downtime* using 90 nodes of *graphene cluster* from *Nancy site* in France of Grid 5 K testbed.

Key Features: The BlobCR framework offers low storage overhead and supports live incremental snapshotting.

Limitations: BlobCR may cause fragmentation and is inapplicable for loosely coupled applications.

4.2.3. BFTCloud

Zhang et al. [61] presented a replication-based fault tolerance framework for byzantine faults in voluntary-resource cloud computing. The framework is capable of successfully completing a user's request in the presence of maximum f faulty nodes in a replication group (BFT group) of size $3f+1$. That is, its fault tolerance capability is approximately 33% as also suggested by Cheraghlou et al. [16]. The problem of finding the number of nodes in a replication group is optimized using the failure probabilities of the available nodes such that the resulting failure probability of the replication group would always be lesser than a threshold failure probability. The framework works in five phases discussed as follows:

- Primary Selection: Each node in cloud is assigned with a QoS value and a priority weight for each of its attributes. The rating of each node is calculated by adding the products of QoS value and priority weight for each attribute. The node with highest rating is then selected as primary. After the primary selection, the cloud module passes the user's request to it.
- Replica Selection: In order to select the required number of replicas, the QoS of each node is observed from the perspective of both cloud module and primary node. The combined QoS is then calculated by applying one of the suitable transformation rules (which can be extended according to the requirement). The rating to each node is then given similar to that in primary selection and the nodes are ordered with respect to their respective ratings. Finally, the required nodes with highest ratings are selected. After the replicas selection, the primary forwards the request for execution to all of them.
- Request Execution: In this phase, all the nodes in BFT group execute the request and respond the cloud module with their respective results. The cloud module then checks for the consistency of obtained results corresponding to four cases discussed later. If results are consistent, next request is forwarded to the primary; else, fault tolerance procedure (primary updating or replica updating) is triggered. The four cases to check the consistency of results are as follows: *Case 1*- If all of $3f+1$ responses are consistent (means no node in BFT is faulty), then the cloud module commits the obtained result to be final. *Case 2*- If $2f+1$ *to* $3f$ results are consistent (means up to f nodes may be faulty), the cloud module sends a commit certificate to all BFT group nodes. If the cloud module receives more than $2f+1$ commit messages as acknowledgement, it commits the obtained consistent results to be final and invokes the replica updating procedure. Else, it resends the commit certificate until it receives more than $2f+1$ acknowledgement messages. *Case 3*- If less than $2f+1$ responses are received (means more than f nodes are faulty), the cloud module resends the request to all the nodes in BFT group. Primary node, as usual, forwards the request to the replicas. Each replica checks whether the sequence numbers of the request sent by the cloud manager and the primary are same or not. If yes, request is executed again the responses are resent to the cloud module. Else, primary updating procedure is triggered. *Case 4*- If more than $2f+1$ responses are received, but lesser than $f+1$ are consistent (means primary forwarded the requests in disturbed order); primary updating procedures is triggered.

- Primary Updating: If a replica suspects the primary to be faulty, it sends a proposal to all other replicas in order to select a new primary. In case at least $f+1$ proposals are receives by a replica, the cloud module selects a new replica from the current replicas in the BFT group.
- Replica Updating: The faulty replicas in the BFT group are removed and new nodes added with respect to the same condition that the failure probability of the group would be lesser than the threshold value.

The performance of the proposed framework is evaluated on Planetlab cloud [62], which consists of 257 computing nodes across 26 countries.

Key Features: BFTCloud is highly reliable fault tolerance framework and is capable to tolerate all byzantine faults.

Limitations: The main drawback is its low resource utilization.

4.2.4. AASIF

Radhakrishnan [63] proposed an adaptive fault tolerant approach to execute the applications in cloud. The applications are modeled using queuing techniques, as continuous functions of time, in terms of network of FIFO servers (considered cloud model) and flow dynamics of requests. At each server, the flow dynamics and server state is computed. On the basis of this computation, further action has to be decided. The incoming requests are first queued according to their arrival times and they are considered as continuous fluid streams. For each stream, there is an algorithmically determined networked path, which it has to flow through. Flow dynamics of each server in the processing server path is computed. Flow dynamics of a server include *server congestion*, *queue length*, *queue growth rate*, *flow processing rate*, and *overall service rate*. After that, one of the *under-saturated*, *saturated*, and *over-saturated* server state in concluded. On the basis of flow dynamics and state of the server, one of the following decisions is made for fault tolerance:

- Redirect the flow to another instance of the server in the cloud.
- Request additional resources at the local node from the cloud infrastructure.
- Spawn additional instances of the server, either at the local node or in another node.
- Combine server instances to conserve resources.

Finally, the server dynamics are calculated which include *mean service rate* and *mean latency*.

Key Features: The AASIF framework guarantees high resource utilization.

Limitations: The fault tolerance processing may be slow due to its adaptive approach.

4.2.5. DAFT

Sun et al. [11] modeled a fault tolerant serviceability in cloud computing environments using the check-pointing technique. The authors first analyzed the mathematical relationship between the different failure rates and check-pointing strategy and then developed a model to provide fault tolerant services in cloud named as DAFT (Dynamic Adaptive Fault Tolerance). The main component of the architecture is the *fault tolerance space* where the measures for fault tolerance are taken. As the name suggests, the check-pointing strategy used is not conventional, but dynamic and adaptive. It means that the check-point interval is not constant and it varies corresponding to *check-pointing overhead*, *fault overhead*, *fault tolerance overhead*, and *failure density*. Whenever the failure rate increases, the system adapts to this condition by dynamically decreasing the check-point interval so that maximum recovery could be done. On the other hand, when the failure rate decreases, the check-point interval increases so that the overhead would be optimized. Besides dynamic and adaptive, the authors integrated the *full check-pointing* and *incremental check-pointing* to realize a *hybrid check-pointing* in order to maintain a trade-off between checkpoint overhead and fault tolerance overhead.

The performance is evaluated in terms of *fault tolerance degree* and *fault tolerance overhead* with respect to different failure rates by simulating DAFT on CloudSim toolkit [57].

Key Features: DAFT is applicable to all crash faults. Its adaptive nature optimizes the overhead.

Limitations: The adaptiveness of DAFT is system based rather than component based, which may affect the fault tolerance.

4.2.6. FTCloud

Zheng et al. [64] proposed a component ranking based fault tolerance framework for cloud applications using different variations of replication technique. The framework follows the *"80-20"* rule given in [65–67] and works in two steps. The first step is about ranking the application components in the cloud, determining their significance, and selecting k most significant components. In the second step, an optimal fault tolerance strategy for each of the k most significant components is selected. Each application is modeled as a directed weighted graph in which each vertex represents an application component possessing a non-negative significance value and each edge is assigned with a weight (depending upon the invocation frequency).

Two algorithms are proposed to rank the application components, viz. *FTCloud1* (considers only the invocation frequencies of the components) and *FTCloud2* (considers both invocation frequencies as well as characteristics of the components). Three variations of replication based fault tolerance are used, viz. *recovery block* [68], *N-version programming* [69], and *parallel* [70]. Failure probability, application cost, and response time of each fault tolerance strategy is computed. The initial step to select the optimal fault tolerance strategy is to exclude those fault tolerance candidates which are unable to satisfy the user constraints. From the refined fault tolerance candidates, the one having least failure probability is selected.

Various algorithms of FTCloud framework are implemented in C++ in order to evaluate the performance in terms of failure probability.

Key Features: The FTCloud is a reliable framework, suitable for software faults due to its high accuracy.

Limitations: It requires complex implementation and has low resource utilization.

4.2.7. CAMAS

Yi et al. [71] proposed five check-pointing and three migration strategies to handle *out-of-bid* situations in spot instances on Amazon cloud. Proposed five check-pointing schemes are: *(a) Optimal Check-pointing* (checkpoint is taken just prior to a probable failure), *(b) Hourly Check-pointing* (checkpoint is taken after each hour since the start of resource usage), *(c) Rising Edge-Driven Check-pointing* (checkpoint is taken every time when the spot price of the currently used resource increases), *(d) Basic Adaptive Check-pointing* (decision is taken every ten minutes whether to take checkpoint or not, depending upon the failure probability), and *(e) Current-Price-Based Adaptive Check-pointing* (same as *basic adaptive check-pointing* except the fact that failure probability is calculated by considering the spot price as well). Similarly, proposed three migration schemes are: *(a) Lowest Price-* The resource instance with lowest current price is chosen for migration with the perception that the probability of *out-of-bid* events in lower price instances is comparatively lesser. *(b) Lowest Failure Rate-* The resource instance with lowest failure rate is chosen for migration with the perception that both user bid and price are considered in failure rate with respect to the probability of failure events. *(c) Highest Failure Rate-* The resource instance with highest failure rate is chosen for migration with the perception that the resources with higher failure rates are more likely to be uncharged for partial hours as per the Amazon's spot pricing rules (refer Fig. 1 of Yi et al. [71]).

For evaluation purpose, 42 types of Amazon's spot instances are

considered. The authors claimed *current-price-based adaptive checkpointing* to be the best in terms of reducing the cost. However, in respect of migration, performances of the all schemes are equivalent.

Key Features: CAMAS is highly adaptive and promises quick fault tolerance with low execution cost.

Limitations: It is confined to tolerate resource contention faults only.

4.2.8. FTM

Jhawar et al. [72] proposed a fault tolerant management framework in IaaS model of cloud computing. In the framework the fault handling is supposed to be done by the third party contracted by the cloud provider and the users are delivered with *fault tolerance as a service* (FTaaS). The fault tolerance is applied at the virtualization layer directly rather than as the application being deployed, by replicating the whole virtual instance. Faults are detected by a run-time monitoring system using heartbeat protocol [25]. The primary component periodically sends a liveness request to all the replicated backups. A timer is maintained for each request. If the replicated backup fails to respond *N* liveness requests within a predefined time, it is considered to be failed. One important feature of FTM is that it implements multiple fault tolerance techniques and a suitable technique is applied as per the user's requirement. The basic components of the architecture are discussed as follows:

- Resource Manager: It is responsible to keep all the information of each resource in cloud, e.g., allocation and release of resources for service provision, avoid resource over provisioning in faulty conditions, etc.
- FTM Kernel: It is the central component of the architecture and it consists of three sub-components: (a) *Service Directory* whose function is to register all the fault tolerance techniques provided by the FTaaS provider. (b) *Composition Engine*- As per the user's requirement, an ordered set of fault tolerance services is provided by the service directory, the *composition engine* generates the optimal solution form the set. (c) *Evaluation Unit*- It continuously validates the fault tolerance solution generated by the composition engine using the protocols corresponding to that solution and does necessary updating if required.
- Client Interface: It provides the users with an interface to the fault tolerance service provider so as to put up their respective requirements.
- Messaging Monitor: It consists of four sub-components: (a) *Replication Manager*- It invokes and manages the execution of replication resources and supervises them to ensure successful operation. (b) *Fault Detection/Prediction Manager*- It is responsible to detect faults in the operational replicas as well as in other resources of the cloud by applying an optimal fault detection algorithm. (c) *Fault Masking Manager*- It masks the occurrence of faults so as to hide them from the users and give them a perception of a failure free operation. (d) *Recovery Manager*- The function of this component is the maintenance and repairing of faulty nodes in the cloud.

Apart from FTM, there are several other frameworks which provide FTaaS, for example [73].

Key Features: The FTM framework works independent of the provider and is applicable to all crash faults.

Limitations: The fault tolerance overhead in relatively high.

4.2.9. FWSS

Poola et al. [74] proposed a fault tolerant algorithm to schedule workflows in cloud. They used the concept of spot and on-demand instances. Spot instance is defined as the idle or unused datacenter capacity. The fault tolerant action of the algorithm takes place to handle premature termination of the spot instances and to handle the performance variations of the cloud resources. Workflows are represented by directed acyclic graphs (DAG) and each node in a DAG represents a task. In order to schedule a task, the slack time (difference between deadline and critical path time) is calculated first. The task is then scheduled on a spot instance. At the occurrence of any fault, the slack time decreases. In such case the algorithm adaptively shifts the execution of the task to an on-demand instance. Spot instances are used in order to improve the resource utilization. To minimize the cost of execution in consequence of a fault, the check-pointing strategy (at a user defined frequency) is used. The FWSS workflow engine consists of five modules, viz. *dispatcher*, *fault tolerance*, *resource allocation*, *runtime estimation*, and *failure estimation*. Dispatcher analyses the data dependencies between the tasks and submits them to the scheduler. Check-pointing of the running tasks is carried in the fault tolerance module. The resource allocation module allocates suitable resources to the tasks. The runtime of a workflow task is estimated in the runtime estimation module using *Downey's Analytical Model* [74]. The failure probabilities of the spot instances are calculated in the failure estimation module.

The algorithm is evaluated on CloudSim toolkit [57] to obtain *execution cost* and *number of failures* w.r.t *deadline time*.

Key Features: FWSS maintains high resource utilization and task completion rate. Moreover; it is specifically suitable for parametric, resource contention, stochastic, and participant faults.

Limitations: FWSS is not applicable to all cloud environments as spot instancing is not a general cloud feature.

4.2.10. MVRS

Zhao et al. [75] proposed a multi-level replication using different types of VM clones, viz. *full clones*, *decoy clones*, and *honeypot clones* for fault tolerance as well as security of mission-critical applications from malicious attacks in cloud computing. Full clones contain the environment to execute the mission-critical applications. To tolerate byzantine faults, voting algorithm is applied over the obtained results. On the other hand, for crash fault tolerance, full clones are synchronized with each other. Moreover, the execution states are periodically checkpointed so as to maintain the synchronization in case a failed clone is regenerated. Decoy clones are installed with such software that could mimic the behavior of full clones and give an illusion to the attackers that the applications are being executed over them. Doing this diverts the attackers and saves the full clones. The idea of decoy clones is drawn from the *chaff cloud* (made of cheap material) spread by military aircraft. Honeypot clones contain the sanitized copy of the application and are generated with the intention that the attackers could interact with them so that their behaviors could be monitored.

The prototypes of the proposed framework are implemented and preliminarily tested on *OpenStack* based private cloud and *Amazon EC2* based public cloud for the proof of proposed concept. One thing to be noted is that the number of each type of clone instances is given by the user rather than by the system.

Key Features: MVRS supports mission critical computing and guarantees application security. Therefore, it is suitable for real time applications.

Limitations: The main disadvantage of MVRS is its high execution overhead.

4.2.11. k-out-of-n FT framework

Chen et al [76] proposed an energy-efficient and fault tolerant framework for data storage and processing in dynamic clouds. They integrated the *k-out-of-n* mechanism [77] from distributed computing into cloud computing. Two functions are developed to store and process data, namely *AllocateData()* and *ProcessData()* respectively. The methodology first separates the storage requests and processing requests and passes them to their respective functions. The probability of operation failure is then estimated, on the basis of which the expected transmission time is computed. After that, the *k-out-of-n* mechanism is applied and the resources are finally allocated. Considering the scope of this paper, only fault tolerance features of the framework are described

here.

The framework consists of five basic components: (a) *Topology Discovery and Monitoring*- As the work targets dynamic clouds, therefore, this component discovers the current topology of the *ad-hoc* network so as to locate the resources. (b) *Failure Probability Estimation*- The failure probability of a node is estimated on the basis of *energy depletion*, *temporary disconnection*, and *application-dependent* factors. (c) *Expected Transmission Time Computation*- It generates a matrix representing the communication cost between any two nodes in the network. (d) *k-out-of-n Allocation*- Data is partitioned into *n* fragments using *erasure code algorithm* [78] and stored in the network such that retrieving *k* fragments would consume minimum energy. (e) *k-out-of-n Processing*- It creates a job consisting of *m* tasks and schedules them on *n* processing nodes such that energy consumption would be minimized. To understand the *k-out-of-n* mechanism, see Fig. 4.

Consider a resource set $R = \{R_0, R_2, \ldots\ldots\ldots R_9\}$ consisting of 10 resources. Suppose, $X = \{R_1, R_2, R_5, R_8, R_9\}$ is a set of $n = 5$ allocated resources with $k = 3$. Let the arrived job $J = (T_1, T_2, T_3)$ is partitioned into three tasks. Now, the three tasks are so replicated on five allocated resources that each subset of X would have at least one instance of each task. Hence, until at least *3-out-of-5* resources work correctly, the execution would go uninterrupted. The framework is simulated by randomly deploying 45 mobile nodes over a network of $400 \times 400\ m^2$.

Key Features: The key features of k-out-of-n framework is its energy efficiency and the capability to tolerate network faults.

Limitations: It's resource utilization is considerably low.

4.2.12. FESTAL

Wang et al. [79] presented a fault tolerant scheduling mechanism for real-time tasks in virtualized clouds. They utilized the primary backup technique for fault tolerance. In the proposed mechanism the users' tasks are queued in an input buffer and then transferred to the scheduler, which has three basic components viz. *resource controller*, *backup copy controller*, and *real-time controller*. At the arrival of a task, *backup copy controller* produces its backup. The *resource controller* then searches for two virtual resources in different hosts those can complete the task before its deadline. If resources are not found, task is rejected. Else, both instances of the task are scheduled on the respective resources. The authors used both active and passive backup schemes adaptively as per the following conditions:

- If expected completion time of the task is lesser than or equal to the task deadline, then passive backup is used.
- If expected completion time of the task is greater than the task deadline, then active backup is used.

Moreover, the authors also introduced the concept of *backup-backup overlapping* (BB Overlapping), which states that the backups of two tasks (t_i and t_j) could overlap with each other on the same virtual machine if their primaries are on different virtual machines. However, it will only happen with the following conditions:

- If host machine of both the tasks is same, they can't be overlapped, regardless of whether their virtual machines are same or different.
- If the backup of t_i is passive and that of t_j is active, they can't be overlapped.
- If the backups of both the tasks are active, they can't be overlapped.
- If the backup of t_i is active, then the backup of t_j can be overlapped if and only if the backup of t_j is passive earliest starting time of backup of t_j is greater than or equal to the addition of expected completion time of primary of t_i and the cancellation time of backup of t_i.

FESTAL is evaluated through simulation experiments using CloudSim toolkit [57] over randomly generated workloads as well as over Google cloud tracelogs.

Key Features: Energy efficiency and high resource utilization are the prime attractions of the FESTAL framework.

Limitations: Execution of tasks will fail if both primary and backup fail simultaneously.

4.2.13. CACS

Cao et al. [73] proposed a service framework for check-pointing the applications in heterogeneous cloud environments. They employed the readymade Distributed Multi-Threaded Check-pointing (DMTCP) package [80] into CACS. DMTCP associates a unique coordinator with each running application, which is responsible for managing the check-pointing of various processes of the application by directly communicating with the DMTCP daemons running on the nodes hosting the application processes.

CACS consists of seven basic working modules: (a) *User API* provides the users with the right to manage their own applications; (b) *Coordinators Database* stores the information regarding all the applications; (c) *Application Manager* supervises application orchestration including failure recovery mechanisms; (d) *Cloud Manager* interacts with cloud infrastructure to manage virtual clusters; (e) *Provision Manager* configures the virtual clusters; (f) *Checkpoint Manager* manages the check-pointing images of the applications; and (g) *Monitoring Manager* is responsible for the detection of possible failures in the application processes.

CACS supports three modes of check-pointing: (a) *User Initiated Check-pointing*- In this mode, the users themselves request the corresponding check-pointing manager to checkpoint their respective running applications; (b) *Periodic Check-pointing*- The running application is check-pointed after a specific period of time; and (c) *Application Initiated Check-pointing*- The running application is check-pointed at the completion of each iteration. In case of fault detection, CACS enables any of the following three recovery mechanisms: (a) *Application Restarting*- the application is restarted since its latest check-pointed state; (b)

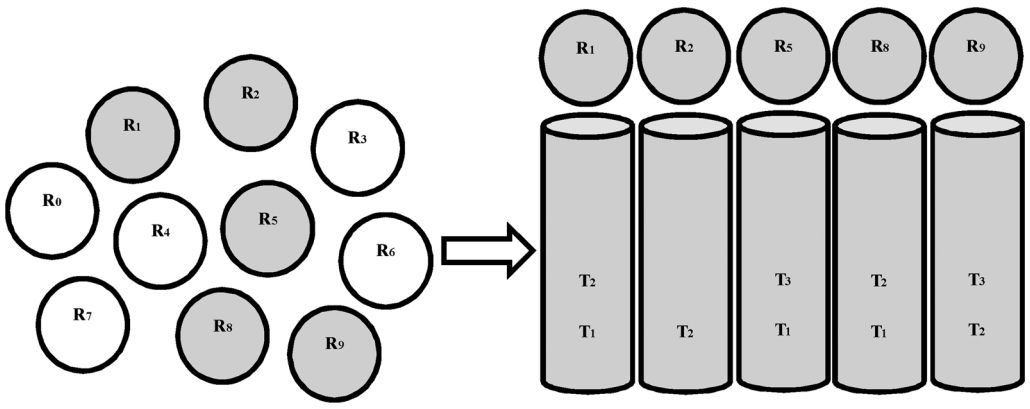

Fig. 4. k-out-of-n mechanism [76].

Application Cloning- A duplicate copy of the original application is created and started from the latest check-pointed state of the original application; and (c) *Application Migration*- The application is terminated at the source cloud and migrated to another cloud.

The performance of CACS is evaluated by conducting experiments on *Snooze* testbed, *OpenStack*, and *Grid 5K* testbed.

Key Features: The CACS framework supports cloud migration.

Limitations: CACS does not have any overhead optimization mechanism.

4.2.14. IRW

Yao et al. [26] proposed an algorithm to reschedule the failed tasks in a workflow being executed in a cloud system. The algorithm is inspired from the immune system of a living being in the context of its fighting capabilities against various diseases. The working architecture of IRW has four modules: (a) *Surveillance Unit*, which is installed in each host and is responsible to detect the faulty VMs in the respective host using heartbeat protocol. If a faulty VM is detected, the (b) *Response Unit* is triggered, which searches the (c) *Memory Unit*, which contains a number of rescheduling strategies. The strategy which is the closest match to the occurred fault is then selected. If no match is found, the (d) *Learning Unit* is triggered, which first searches for a suitable VM in the closest (squared Euclidean distance) cluster. If a suitable VM is found, failed task is rescheduled on it and this VM is updated in the *memory unit*. If no VM is found, a new VM is created on an active host which meets the requirement of new VM, failed task is rescheduled on it, and the *memory unit* is updated. If no active host meets the requirement, a host in sleep mode is turned on to create a new VM in order to reschedule the failed task, and the *memory unit* is updated. The performance of IRW algorithm is evaluated through simulation on four real world workflows, viz. *Montage*, *Epigenomics*, *CyberShake*, and *Inspiral* as well on a few randomly generated workflows.

Key Features: IRW promises high task completion rate and low makespan delay.

Limitations: IRW is not suitable for hard deadline applications.

4.2.15. SAFTP

Chen and Jiang [41] proposed an adaptive fault tolerant strategy for cloud application, named as SelfAdaptionFTPlace. The proposed strategy works in two phases. In the first phase, different fault tolerance methods are sorted as per the user's provided constraints. Four fault tolerance methods are considered in the work viz. *retry*, *recovery block*, *N-version programming*, and *active*. Three user provided constraints are considered viz. *response time*, *failure rate*, and *resource consumption*. The best fault tolerance method is then chosen in the context of user provided constraint. In the second phase the virtual machines are placed as per the selected method. Different modules of the SAFTP prototype implementation are described as follows:

- Application requirement initialization: Users' applications in cloud are initialized corresponding to the respective users' constron: Users' applications in cloud are initialized corresponding to the respective usersaints.
- Fault tolerance strategy selection: Fault tolerance methods are sorted and best method is selected as per the user's constraints.
- VM placement model transformation: Virtual machines assigned to the application are then placed according to the selected fault tolerance method.

Key Features: SAFTP is an adaptive framework which allows users' interaction for fault tolerance.

Limitations: Not suitable for inexperienced users.

4.2.16. SAFTP

Ding et al. [81] proposed a fault tolerant workflow scheduling algorithm using *primary-backup* and *job-migration* methods. The cloud architecture and the execution flow of each task in the workflows are similar to those in FESTAL [79]. Each workflow is considered as a DAG, where the vertices represent the tasks and the edges represent the dependencies between the tasks. Instead of considering a single deadline for a complete workflow, the authors divide the workflow deadline into tasks' deadlines depending upon the number of tasks and the size of each task. Each task t_j in the workflow has two copies viz. *primary* (t_j^P) and *backup* (t_j^B). If the primary fails, execution continues at the backup. However, there are certain cases to create the backup of a task, which are

- t_j^B cannot be placed on the host where t_j^P is placed.
- t_j^B cannot be placed on the hosts where the predecessors of t_j^P are placed.
- t_j^B cannot start execution until all the backups of direct predecessors of t_j finishes and transferred their data to t_j^B.

Furthermore, if the backup of a task (t_j^B) fails, it is migrated to some other host, but not to any of the hosts where the predecessors of t_j are placed. The performance of FTESW algorithm is evaluated through simulation on four real world workflows, viz. *Montage*, *Epigenomics*, *CyberShake*, and *Inspiral* as well as on a few randomly generated workflows similar to IRW [26].

Key Features: The FTESW framework offers high resource utilization.

Limitations: Workflow execution will interrupt if both primary and backup fail simultaneously.

Apart from the above proactive and reactive frameworks, Amoon [82] presented a hybrid fault tolerance framework *HFFC* in cloud computing. The framework is called hybrid as it uses both proactive and reactive approaches. The main component of the framework is the *allocator* which contains three modules, viz. *QoS controller*, *VMs database*, and *broker*. QoS controller receives user's request along with QoS requirements and queries the VMs database whether suitable VMs could be available or not. The VMs database (which contains all the information about VMs in cloud) replies the QoS controller. If suitable VMs are available, QoS controller accepts the user request and passes it to the broker. If no suitable VM is available, request is rejected. Broker is the most important module which further contains three sub-modules, viz. *VMFT (virtual machine fault tolerance) selection*, *VMs classifier*, and *dispatcher*. The VMFT selection module uses the Fault Tolerance Strategy Selection (FTSS) algorithm to select the most suitable fault tolerance strategy for the user's request. The VMs classifier uses Virtual Machine Classification (VMC) algorithm to classify the resources with can fulfill the user's request. The FTSS and VMC algorithms are explained later in this section. The dispatcher module finally dispatches the selected VMs to execute the request. The FTSS algorithm takes *application cost* and *deadline time* as the input from the customer. Then for each VM, it calculates the estimated cost and time if the application would be executed on it and shortlists *m* most valuable VMs in the cloud using VMC algorithm. For each VM in the *m* list, it is computed whether it could execute the application within the application cost and deadline time. If no VM is found, application is discarded. If more than one VMs are found, replication strategy is used; else, check-pointing strategy is used. The VMC algorithm classifies the VMs according to the time for which they have been used and their failure probabilities. The failure probability of a VM is obtained using *Poisson distribution*. The value of a VM is calculated in terms of a *selection parameter*, which is equal to the product of its *usage time* and *failure probability*. The more the value of selection parameter, the more valuable is the VM.

Amoon [83] proposed another fault tolerance framework for cloud environment. The framework is adaptive as it implements either replication or checkpoint-restart methods according to the users' requests and VMs status. The system model of AFRCE is similar to that of HFFC. Similar to FTSS algorithm in HFFC, it implements a Selecting Fault Tolerance (SFT) algorithm, which selects either replication or checkpoint-restart. According to SFT algorithm, if the number of VMs which

can carry out the request is more than 1, replication is selected. Else, checkpoint-restart is selected. In case of replication, AFRCE targets the challenge of adjusting the number of replicas by calculating the values of VMs. The value of a VM is a function of its failure probability and the profit of its use. On the other hand, if checkpoint-restart is selected, AFRCE targets the challenge of adjusting the checkpoint frequency. The checkpoint frequency is also adjusted with respect to the failure probability. The framework is evaluated using CloudSim simulator [57] in terms of *throughput*, *availability*, *time overhead*, and *monetary waste overhead*.

5. Discussion

In the survey, we observe how much a particular fault category is focused upon and how much a particular fault tolerance method is utilized in the existing research. Estimation is drawn in from more than 40 frameworks included in the prominent surveys already published in literature [17–19,16,20,84,21] along with the frameworks discussed in the present survey. Fig. 5 gives the applicability graphs of fault tolerance methods, showing the prevalence of reactive approaches (with 86% applicability) over proactive approaches.

It is conclusive that the proactive methods need consistent monitoring of the system. They highly rely on *learning and prediction* using *probability theory* and *artificial intelligence*. The tasks executed under proactive fault tolerance remain uninterrupted until the system behaves according to the probability of the system's future state. However, in case of any deviation in system behavior or any inaccurate prediction, these methods become ineffective. It may result in severe consequences, and might lead the system to unworkable state. For example; ASSURE [44] and SHelp [43] are confined to tolerate software faults, while FTDG [54] can tolerate parametric faults only. In case any other fault type occurs, they would not be able to predict accurately might become ineffective. The applicability of WSRC [37] is confined to infrastructure layer only and the fault occurred in the above layers cannot be predicted. Moreover, the overhead of monitoring the system consistently or even periodically is also a considerable factor. In *preemptive migration*, the total fault tolerance overhead includes system monitoring as well as task migration from the faulty predicted node. In *system rejuvenation*, the periodic backup of the system causes considerable storage overhead. However, as discussed in section 3, backing-up period can be varied as is done in [37,53] so that overhead could be controlled. Now, in case of *self-healing* method, if a condition (which has never been experienced by the system before) occurs, it will be required to develop and deploy new recovery procedures.

On the other hand, reactive approaches are mostly represented by *replication* method (36% applicability), while *checkpoint restart* (29% applicability) and *job migration* (21% applicability) generally work as auxiliaries. [83] also suggests that replication is the method used by most of the current cloud systems. *Replication* method takes the advantage of high resource availability in cloud, but it consumes a lot of resources and contributes towards the total execution cost. If the fault tolerance is adaptive (e.g. HHFC, AFRCE, etc.) [82] or uses multiple methods (e.g. FTM) [72], the execution cost climbs higher. To avoid the resource wastage and to utilize the execution cost in replication, *passive replication* was introduced. Authors in [85,2,86] suggest that the backup resources in passive replication could be utilized by executing low priority secondary tasks on them. Another issue with replication is to calculate optimal group size. Many researchers calculated the replication group size on the basis of various resources' parameters, e.g. *reliability*, *trust and reputation*, *failure rate*, etc. For inclusion in this paper; [85,2,86] calculate the group size w.r.t resource reliability, Zhang et al. [61] calculate w.r.t QoS of the resources, Zheng et al. [64] calculate w.r.t invocation frequency and characteristics of resources, [87] calculates w.r.t failure rates of the resources. Besides this, [88] state that a high degree of fault tolerance could be achieved if a data item is replicated across three servers, provided that failures of those servers are mutually independent. Similar to that, [89] propose limited duplication for dependent tasks' scheduling. In addition, [55] also obtained the results using three replicas. FESTAL [79] and FTESW [81] introduced the concept of backup overlapping to avoid resource wastage in replication. The case with *checkpoint restart* is very much similar to *system rejuvenation* in the sense that the overhead can be optimized by varying the checkpoint interval as done by the authors in [11,71]. Moreover, *shadowing* is another technique to optimize the overhead as done in [58].

Furthermore, Fig. 6 shows how much a particular fault category (crash or byzantine) is targeted in the literature.

It can be noticed that 43% frameworks target crash faults, 26% target byzantine faults, while 31% target both. It shows that researchers are slightly more motivated towards crash fault tolerance. It may be due to the reason that the detection and tolerance of crash faults seems easier to simulate in comparison to those of byzantine faults. However, there may be other reasons as well. A comparative analysis of the frameworks surveyed in section 4 is given in Table 3.

6. Research directions

Cloud computing is a major attraction in industry, commerce, education, and IT professionals. Over the years its user base has consistently widened. To meet the cloud service requirements of the vast user base, numerous cloud service providers have appeared. Every day millions of tasks are emerging for execution in the cloud environment

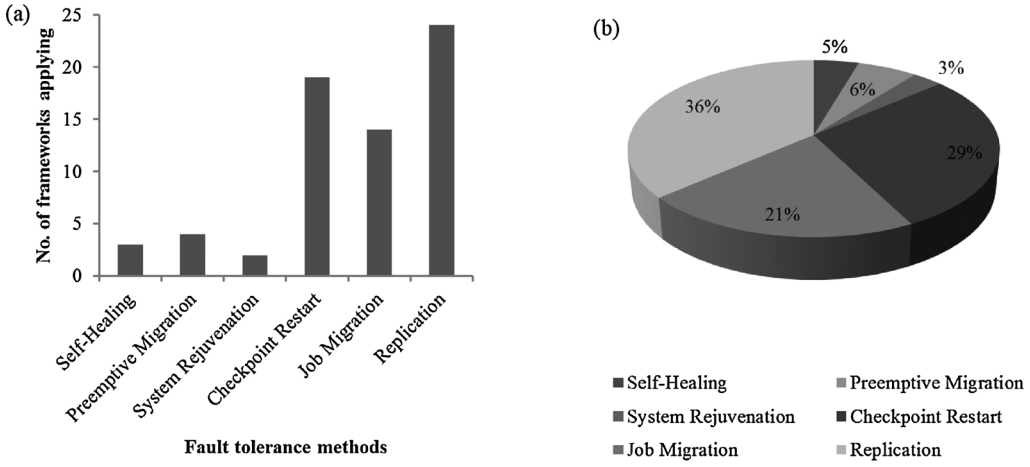

Fig. 5. (a) Applicability count and (b) Applicability percentage of fault tolerance methods.

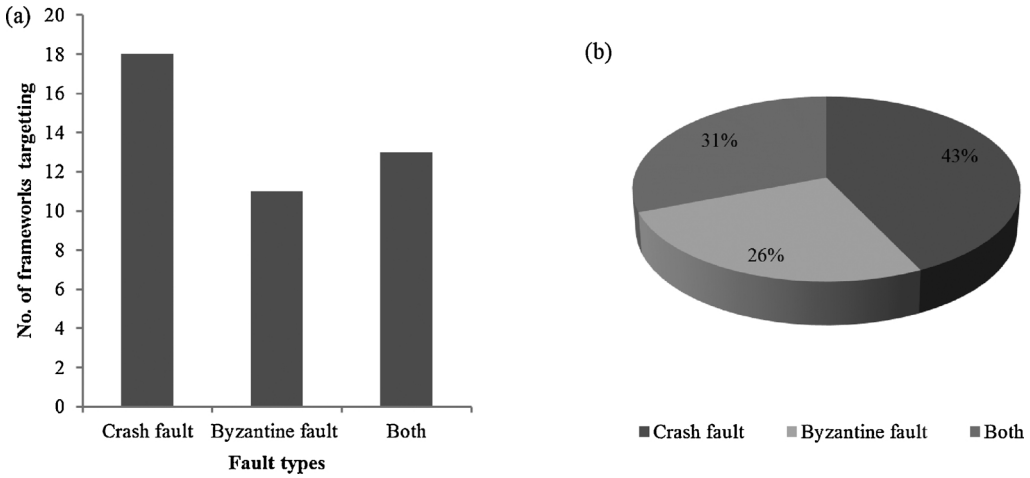

Fig. 6. (a) No. of frameworks and (b) Percentage of frameworks targeting different fault types.

[90]. For instance, 25 million tasks were submitted to Google cloud in the span of 29 days [79]. Service reliability is one of the primary characteristic of cloud computing and therefore it successfully achieves a remarkable service completion rate. Wang et al. [79] and Ding et al. [81] claimed to achieve about 95% task completion rate on the real-world workload. However, 5% task failure rate results in the failure of thousands of tasks, which cannot be ignored. Therefore, highly efficient fault tolerance frameworks are required to minimize the task failure rate.

Though reactive fault tolerance methods are prevalent among researchers till now but due to the ongoing advancements in machine learning, the devices are becoming smarter and artificially intelligent which is increasing the scope of research in proactive fault tolerance methods (specifically *self-healing*). Overhead optimization is further one of the open research area in cloud computing. It can considerably enhance the applicability of proactive methods among both researchers as well as professionals. Applications in cloud demand high serviceability and reliability. Therefore, it is difficult to design a fault tolerance framework (using a particular fault tolerance method), which is both highly reliable and optimal. However, by combining two or more fault tolerance methods to form a hybrid fault tolerance framework may solve the problem. Certain frameworks (both proactive and reactive) in the literature applied *checkpoint restart* as the auxiliary method to invoke fault tolerance. However, it has been observed that *preemptive migration* could also be a suitable option to be applied as auxiliary. *Replication* is generally used for storage applications to achieve zero downtime. For computational applications, *passive replication* is accompanied by *checkpoint restart* to reduce the computational downtime, but practically it is almost impossible to achieve zero computational downtime using *checkpoint restart*. However, using *preemptive migration* instead of *checkpoint restart* along with *passive replication* may do the job.

Furthermore, the existing frameworks counter only a few fault types and to the best of our knowledge no framework has claimed to be enough fault tolerant to handle each fault type. From the research perspective, again due to the advancements in machine learning, novel fault tolerance frameworks (smart enough to apply different strategies at different situations) are expected to counter all different fault types. In addition, the existing fault tolerance techniques proposed in literature are broadly service provider centric. The consumer is provided with the readymade fault tolerance mechanisms implemented at the service provider end. Cloud is a pay-per-use service infrastructure where the customers pay only for the service(s) they use. At the time, when a customer can select the required service from cloud under the pay-per-use service model, he/she should also be provided the flexibility to decide the service level guarantee in terms of fault tolerance. Therefore, a mechanism of service orchestration in cloud which includes service selection as well as price oriented flexible fault tolerance is needed.

Fault tolerance is a combined action of *fault forecasting* and *fault prevention* (proactive) or *fault detection* and *fault recovery* (reactive). Most of the frameworks proposed in literature focus on the second aspect i.e., fault prevention and fault recovery. Commercially, cloud datacenters contain thousands of physical hosts with different reliabilities [91,10]. It shows how complex it is to forecast/detect faults and failures in such a gigantic network. Hence, frameworks capable of providing complete fault tolerance (forecast/detection + prevention/recovery) would be highly appreciated.

In the present IT systems, fault tolerance provisioning generally focuses on the hardware infrastructure. It means that the hardware infrastructure is more susceptible to faults than the software infrastructure. Unbalanced workload distribution may be a vital reason for this, which causes hosts' overloading and consequently their failure. In this regard, fault tolerance can be achieved through efficient workload distribution using clustering [92,93] and load balancing [94]; [95]. Clustering is a technique to form the groups of similar objects. In a distributed network, clusters of tasks and resources can be formed to perform efficient task-resource mapping. Abdulhamid et al. [92] proposed clustering-based fault tolerance scheduling algorithm for autonomous tasks in cloud. Chen et al. [93] proposed three task clustering algorithms in workflows to provide fault tolerance. On the other hand, load balancing in cloud has also been an issue of attraction among researchers due to its far reaching consequences in the resource usage optimization [96] and energy-efficiency [97]. As already discussed, overloading causes hosts' failure, therefore load balancing can have great application in fault tolerance as well. Idris et al [94] proposed a fault tolerance algorithm for job scheduling in grid computing environment. Fault tolerance is implemented through load balancing using ant colony optimization. For cloud environment, Moghtadaeipour and Tavoli [95] proposed a load balancing architecture using fuzzy logic for energy optimization as well as fault tolerance. A limited research is attributed to the application of load balancing in fault tolerance in cloud and needed further explorations. Based on the above discussion, following probable areas are recommended for future research:

- Towards the improvement of task completion rate in cloud.
- Overhead optimization of proactive fault tolerance methods.
- Provision of hybrid fault tolerance in cloud.
- Application of preemptive migration as auxiliary fault tolerance method.
- Provision of fault tolerance to counter each fault type in cloud.

Table 3
Comparison of different fault tolerance frameworks in cloud.

Framework	Basic Approach (Proactive/ Reactive)	Method(s) Used	Application Scenario	Key Features/Advantages	Limitations/Disadvantages
ASSURE	Proactive	Self-healing, checkpoint restart	Software faults	Works without base OS kernel changes	May be slow at times
SHelp	Proactive	Self-healing, checkpoint restart	Software faults	Faster functioning and lesser overhead	Tolerate limited fault types
AFTRC	Reactive	Replication, checkpoint restart	Configuration and parametric faults	Accuracy, real-time applicability	Limited experimentation
BlobCR (2011)	Reactive	Checkpoint restart	Hardware faults	Low storage overhead	Inapplicable for loosely coupled applications
BlobCR (2013)	Reactive	Checkpoint restart	Hardware faults	Live incremental snapshotting	Fragmentation may occur
BFTCloud	Reactive	Replication	Byzantine Faults	High reliability	Low resource utilization
PFHX	Proactive	Preemptive migration	Hardware faults	Supports live migration	High execution cost
PFHC	Proactive	Preemptive migration	Hardware faults	Lower execution cost than PFHX	Complex implementation
AASIF	Reactive	Job migration, replication	Resource contention	High resource utilization	Fault tolerance action may be slow
DAFT	Reactive	Checkpoint restart	Crash faults	Adaptive, optimized overhead	Adaptiveness is system-based
FTCloud1	Reactive	Replication	Software faults	High reliability	Low resource utilization
FTCloud2	Reactive	Replication	Software faults	More accuracy	Complex implementation
CAMAS	Reactive	Checkpoint restart, job migration	Resource contention faults	Fast, low execution cost	Tolerate limited fault types
FTM	Reactive	All reactive methods	Crash faults	Fault tolerance as a service	High overhead and complex implementation
WSRC	Proactive	System rejuvenation	Software faults	High availability, optimized overhead	Applicable at infrastructure layer only
FWSS	Reactive	Checkpoint restart, job migration	Parametric, resource contention, stochastic, and participant faults	High resource utilization	Limited applicability
MVRS	Reactive	Checkpoint restart, replication	General crash and byzantine faults, malicious attacks	Secure, supports mission critical computing	High overhead
k-out-of-n	Reactive	Replication	Hardware and network faults	Energy efficient	Low resource utilization
FESTAL	Reactive	Replication	Host failures	High resource utilization	Inapplicable if primary & backup fail simultaneously
SRFSC	Proactive	System rejuvenation	Software aging and crashes	High availability, supports multiple rejuvenations of independent VMs	Limited failure aspects considered
CACS	Reactive	Checkpoint restart, job migration	VM crashes, unhealthy applications	Supports cloud migration	No overhead optimization
HFFC	Hybrid	Checkpoint restart, replication	Configuration and constraint faults	Economical	Slow functioning
AFRCE	Reactive	Checkpoint restart, replication	General faults	Adaptive	Low throughput
IRW	Reactive	Job rescheduling	VM crashes	Low makespan, high task completion rate	Not suitable for hard deadline applications
SAFTP	Reactive	Replication	Software faults	Adaptive	Limited experimentation
IVFS	Reactive	Checkpoint restart, replication	Host failures and software faults	Low checkpoint overhead	Limited experimentation
FTDG	Proactive	Pre-emptive migration	Parametric faults	Low response time	Limited fault applicability
FTESW	Reactive	Job migration, replication	Host failures	High resource utilization	Inapplicable if primary & backup fail simultaneously

- Provision of fault tolerance through clustering and load balancing.
- Towards price flexible fault tolerance in cloud.
- Provision of efficient fault forecasting and detection mechanisms.

7. Conclusions

Fault tolerance has been one of the major issues in cloud computing environments. Dynamic infrastructure and complex configuration are among the key reasons. In this paper, different fault types (along with their causes) and various fault tolerance approaches in cloud computing have been discussed in a systematic manner. Eminent fault tolerance frameworks have also been surveyed in terms of their basic approach, fault applicability and key features. Following conclusions have been made:

- Researchers are more motivated towards addressing crash faults rather than byzantine faults.
- Reactive fault tolerance methods are more often applied rather than proactive ones.
- Higher overheads and complex implementations are observed as the core reasons behind the reluctance of researchers towards proactive approaches.
- Replication is the most applied fault tolerance technique followed by checkpoint restart and job migration respectively.
- In many frameworks, checkpoint restart and job migration are used as auxiliary techniques with replication.

The paper includes a comparative analysis of different fault tolerance frameworks which will facilitate the researchers to select the framework of their interest. For example, a researcher wants to work upon reactive methods using replication may look through *AFTRC*, *BFTCloud*, *FTM*, *k-out-of-n*, and *FESTAL* etc. Furthermore, if one needs to look upon the frameworks dealing with resource contention faults, he/she may opt for *AASIF* and *CAMAS*. In addition, the survey also enlists various challenges regarding the implementation of fault tolerance in cloud and shows the possible future directions for research in this context.

Acknowledgements

This research work is supported by Sant Longowal Institute of Engineering and Technology, Sangrur, India under research fellowship scheme.

We are thankful to the journal editor and anonymous reviewers for their valuable comments to finalize the survey.

References

[1] P. Mell, T. Grance, The NIST definition of cloud computing [WWW document], Natl. Inst. Stand. Technol. (2011) URL http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (Accessed 9 January 2016).
[2] M. Hasan, M.S. Goraya, Priority based cooperative computing in cloud using task backfilling, Lect. Notes Softw. Eng. 4 (2016) 229–233, http://dx.doi.org/10.18178/lnse.2016.4.3.255.
[3] G.A. Lewis, Role of standards in cloud-computing interoperability, Proceedings of the Annual Hawaii International Conference on System Sciences (2013) 1652–1661, http://dx.doi.org/10.1109/HICSS.2013.470.
[4] RightScale, State of the Cloud Report [WWW Document], RightScale, 2016 URL http://assets.rightscale.com/uploads/pdfs/RightScale-2016-State-of-the-Cloud-Report.pdf (Accessed 11 February 2016).
[5] B. Kazarian, D. Baron, B. Hanlon, R. Tully, J. Fjeldal, SMB Cloud Adoption Study [WWW Document]. Edge Strateg, (2012) URL http://www.edgestrategies.com/component/k2/item/117-just-released-2012-microsoft-edge-technologies-smb-cloud-adoption-study.html (Accessed 11 February 2016).
[6] M. Tebaa, S. El Hajji, From single to multi-clouds computing privacy and fault tolerance, Proceedings International Conference on Future Information Engineering (2014) 112–118, http://dx.doi.org/10.1016/j.ieri.2014.09.099 (Elsevier B.V.).
[7] A. Abid, M.T. Khemakhem, S. Marzouk, M. Ben Jemaa, T. Monteil, K. Drira, Toward antifragile cloud computing infrastructures, Procedia Comput. Sci. 32 (2014) 850–855, http://dx.doi.org/10.1016/j.procs.2014.05.501.
[8] G. Fagg, J. Dongarra, FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world, Recent Advances in Parallel Virtual Machine and Message Passing Interface, (2000), pp. 1–8, http://dx.doi.org/10.1007/3-540-45255-9_47.
[9] X. Lin, A. Mamat, Y. Lu, J. Deogun, S. Goddard, Real-time scheduling of divisible loads in cluster computing environments, J. Parallel Distrib. Comput. 70 (2010) 296–308, http://dx.doi.org/10.1016/j.jpdc.2009.11.009.
[10] R. Jhawar, V. Piuri, Fault tolerance and resilience in cloud computing environments, in: J. Vacca (Ed.), Computer and Informaion Security Handbook, 2013, pp. 1–29, , http://dx.doi.org/10.1109/CLOUD.2011.16.
[11] D. Sun, G. Chang, C. Miao, X. Wang, Modelling and evaluating a high serviceability fault tolerance strategy in cloud computing environments, Int. J. Secur. Netw. 7 (2012) 196–210, http://dx.doi.org/10.1504/IJSN.2012.053458.
[12] A. Tchernykh, U. Schwiegelsohn, V. Alexandrov, E. Talbi, Towards understanding uncertainty in cloud computing resource provisioning, Proceedings Internatinal Conference on Computational Science (2015) 1772–1781, http://dx.doi.org/10.1016/j.procs.2015.05.387.
[13] T. Wang, W. Zhang, C. Ye, J. Wei, H. Zhong, T. Huang, FD4C: automatic fault diagnosis framework for web applications in cloud computing, IEEE Trans. Syst. Man Cybern. Syst. 46 (2016) 61–75, http://dx.doi.org/10.1109/TSMC.2015.2430834.
[14] W. Ahmed, Y.W. Wu, A survey on reliability in distributed systems, J. Comput. Syst. Sci. 79 (2013) 1243–1255, http://dx.doi.org/10.1016/j.jcss.2013.02.006.
[15] S. Hernández, J. Fabra, P. Álvarez, J. Ezpeleta, Using cloud-based resources to improve availability and reliability in a scientific workflow execution framework, The Fourth International Conference on Cloud Computing, GRIDs and Virtualization (2013) 230–237.
[16] M.N. Cheraghlou, A. Khadem-Zadeh, M. Haghparast, A survey of fault tolerance architecture in cloud computing, J. Netw. Comput. Appl. 61 (2016) 81–92, http://dx.doi.org/10.1016/j.jnca.2015.10.004.
[17] H. Agarwal, A. Sharma, A comprehensive survey of fault tolerance techniques in cloud Computing, 2015 Intl. Conference on Computing and Network Communications (CoCoNet'15) (2015) 408–413, http://dx.doi.org/10.1109/CoCoNet.2015.7411218.
[18] Z. Amin, N. Sethi, H. Singh, Review on fault tolerance techniques in cloud computing, Int. J. Comput. Appl. 116 (2015) 11–17.
[19] S.M.A. Ataallah, S.M. Nassar, E.E. Hemayed, Fault tolerance in cloud computing – Survey, 11th International Computer Engineering Conference (2015) 241–245.
[20] L.P. Saikia, Y.L. Devi, Fault tolererance techniques and algorithms in cloud system, Int. J. Comput. Sci. Commun. Netw. 4 (2014) 1–8.
[21] A. Tchana, L. Broto, D. Hagimont, Fault tolerant approaches in cloud computing infrastructures, The Eighth International Conference on Autonomic and Autonomous Systems (2012) 42–48.
[22] D. Oppenheimer, A. Ganapathi, D.A. Patterson, Why do internet services fail, and what can be done about it? USENIX Symposium on Internet Technologies and Systems (2003) 1–15.
[23] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, A view of cloud computing, Commun. ACM 53 (2010) 50–58, http://dx.doi.org/10.1145/1721654.1721672.
[24] M. Ali, S.U. Khan, A.V. Vasilakos, Security in cloud computing: opportunities and challenges, Inf. Sci. (Ny) 305 (2015) 357–383, http://dx.doi.org/10.1016/j.ins.2015.01.025.
[25] J. Dong, Z. Decheng, L. Hongwei, Y. Xiaozong, DPHM: a fault detection protocol based on heartbeat of multiple master-nodes, J. Electron. 24 (2007) 544–549, http://dx.doi.org/10.1007/s11767-006-0122-5.
[26] G. Yao, Y. Ding, L. Ren, K. Hao, L. Chen, An immune system-inspired rescheduling algorithm for workflow in cloud systems, Knowl.-Based Syst. 99 (2016) 39–50, http://dx.doi.org/10.1016/j.knosys.2016.01.037.
[27] A. Bala, I. Chana, Fault tolerance- challenges, techniques and implementation in cloud computing, Int. J. Comput. Sci. 9 (2012) 288–293.
[28] W. Qiu, Z. Zheng, X. Wang, X. Yang, M.R. Lyu, Reliability-based design optimization for cloud migration, IEEE Trans. Serv. Comput. 7 (2014) 223–236, http://dx.doi.org/10.1109/TSC.2013.38.
[29] T. Zaidi, Modeling for fault tolerance in cloud computing environment, J. Comput. Sci. Appl. 4 (2016) 9–13, http://dx.doi.org/10.12691/jcsa-4-1-2.
[30] C. Engelmann, G.R. Vallée, T. Naughton, S.L. Scott, Proactive fault tolerance using preemptive migration, Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009 (2009) 252–257, http://dx.doi.org/10.1109/PDP.2009.31.
[31] R. Salvador, A. Otero, J. Mora, E.D. Torre, L. La Sekanina, T. Riesgo, Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems, Proceedings International Conference on Reconfigurable Computing and FPGAs (2011) 164–169, http://dx.doi.org/10.1109/ReConFig.2011.37.
[32] D. Ghosh, R. Sharman, H. Raghav Rao, S. Upadhyaya, Self-healing systems – survey and synthesis, Decis. Support Syst. 42 (2007) 2164–2185, http://dx.doi.org/10.1016/j.dss.2006.06.011.
[33] W. Haque, A. Aravind, B. Reddy, Pairwise sequence alignment algorithms – a survey, Proceedings of the 2009 Conference on Information Science, Technology and Application (2009) 96–103, http://dx.doi.org/10.1145/1551950.1551980.
[34] H. Tu, Comparisons of self-healing fault-tolerant computing schemes, Proceedings World Congress on Engineering and Computer Science (2010) 1–6.
[35] T.-H. Lai, S. Sahni, Preemptive scheduling of a multiprocessor system with memories to minimize maximum lateness, SIAM J. Comput. 13 (1984) 690–704.
[36] A. Polze, P. Tröger, F. Salfner, Timely virtual machine migration for pro-active fault tolerance, Proceedings 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (2011) 234–243, http://dx.doi.org/10.1109/ISORCW.2011.42.
[37] D. Bruneo, S. Distefano, F. Longo, A. Puliafito, M. Scarpa, Workload-based software rejuvenation in cloud systems, IEEE Trans. Comput. 62 (2013) 1072–1085, http://

dx.doi.org/10.1109/TC.2013.30.
[38] B. Nicolae, F. Cappello, BlobCR: virtual disk based checkpoint-restart for HPC applications on IaaS clouds, J. Parallel Distrib. Comput. 73 (2013) 698–711, http://dx.doi.org/10.1016/j.jpdc.2013.01.013.
[39] G. Bosilca, R. Delmas, J. Dongarra, J. Langou, Algorithm-based fault tolerance applied to high performance computing, J. Parallel Distrib. Comput. 69 (2009) 410–416, http://dx.doi.org/10.1016/j.jpdc.2008.12.002.
[40] N. Naksinehaboon, M. Paun, R. Nassar, B. Leangsuksun, S. Scott, High performance computing systems with various checkpointing schemes, Int. J. Comput. Commun. Control 4 (2009) 386–400.
[41] X. Chen, J. Jiang, A method of virtual machine placement for fault- tolerant cloud applications, Intell. Autom. Soft Comput. (2016) 1–11, http://dx.doi.org/10.1080/10798587.2016.1152775.
[42] W. Zhao, P.M. Melliar-Smith, L.E. Moser, Fault tolerance middleware for cloud computing, Proceedings IEEE 3rd International Conference on Cloud Computing (2010) 67–74, http://dx.doi.org/10.1109/CLOUD.2010.26.
[43] G. Chen, H. Jin, D. Zou, B.B. Zhou, W. Qiang, G. Hu, SHelp: automatic self-healing for multiple application instances in a virtual machine environment, Proceedings – IEEE International Conference on Cluster Computing ICCC (2010) 97–106, http://dx.doi.org/10.1109/CLUSTER.2010.18.
[44] S. Sidiroglou, O. Laadan, C.R. Perez, N. Viennot, J. Nieh, A.D. Keromytis, ASSURE: automatic software self-healing using REscue points, Proceedings Architectural Support for Programming Languages and Operating Systems (2009) 37–48, http://dx.doi.org/10.1145/1508244.1508250.
[45] I.P. Egwutuoha, S. Chen, D. Levy, B. Selic, R. Calvo, A proactive fault tolerance approach to high performance computing (HPC) in the cloud, Second International Conference on Cloud and Green Computing (2012) 268–273, http://dx.doi.org/10.1109/CGC.2012.22.
[46] R. Arvind, A. Vinnarsi, Temperature monitoring with the linux kernel on a multi core processor, Int. J. Innov. Res. Sci. Eng. Technol. 4 (2015) 876–883, http://dx.doi.org/10.15680/IJIRSET.2015.0403011.
[47] K. Toshniwal, J.M. Conrad, A Web-based sensor monitoring system on a linux-based single board computer platform, Conference Proceedings – IEEE SOUTHEASTCON (2010) 371–374, http://dx.doi.org/10.1109/SECON.2010.5453851.
[48] A.B. Nagarajan, F. Mueller, C. Engelmann, S.L. Scott, Proactive fault tolerance for HPC with Xen virtualization, Proceedings of the 21st Annual International Conference on Supercomputing ICS 07 (2007) 23–32, http://dx.doi.org/10.1145/1274971.1274978.
[49] T. Fukai, Y. Omote, T. Shinagawa, K. Kato, OS-independent live migration scheme for bare-metal clouds, Proceedings IEEE/ACM 8th International Conference on Utility and Cloud Computing (2015) 80–89, http://dx.doi.org/10.1109/UCC.2015.23.
[50] P. Rad, A.T. Chronopoulos, P. Lama, P. Madduri, C. Loader, Benchmarking bare metal cloud servers for HPC applications, Proceedings IEEE International Conference on Cloud Computing in Emerging Markets (2015) 153–159, http://dx.doi.org/10.1109/CCEM.2015.13.
[51] P. Jorgensen, D. Pauksztello, Classification of co-slicings and co-t-structures for the kronecker algebra, J. Pure Appl. Algebr. 219 (2015) 569–590, http://dx.doi.org/10.1016/j.jpaa.2014.05.015.
[52] A. Bobbio, A. Puliafito, M. Scarpa, M. Telek, WebSPN: a WEB-accessible petri net tool, International Conference on Web-Based Modeling & Simulation (1998) 137–142.
[53] J. Liu, J. Zhou, R. Buyya, Software rejuvenation based fault tolerance scheme for cloud applications, Proceedings IEEE 8th International Conference on Cloud Computing (2015) 1115–1118, http://dx.doi.org/10.1109/CLOUD.2015.164.
[54] D. Sun, G. Zhang, C. Wu, K. Li, W. Zheng, Building a fault tolerant framework with deadline guarantee in big data stream computing environments, J. Comput. Syst. Sci. (2017) 1–20, http://dx.doi.org/10.1016/j.jcss.2016.10.010.
[55] S. Malik, F. Huet, Adaptive fault tolerance in real time cloud computing, Proceedings — IEEE World Congress on Services (2011) 280–287, http://dx.doi.org/10.1109/SERVICES.2011.108.
[56] B. Mohammed, M. Kiran, I. Awan, K.M. Maiyama, An integrated virtualized strategy for fault tolerance in cloud computing environment, International IEEE Conference on Ubiquitous Intelligence & Computing (2016) 542–549, http://dx.doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.158.
[57] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.F. De Rose, R. Buyya, CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw. – Pract. Exp. 41 (2011) 23–50, http://dx.doi.org/10.1002/spe.995.
[58] B. Nicolae, F. Cappello, BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots, Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (2011) 1–12, http://dx.doi.org/10.1145/2063384.2063429.
[59] B. Nicolae, G. Antoniu, L. Boug, D. Moise, A. Carpen-Amarie, BlobSeer: next-generation data management for large scale infrastructures, J. Parallel Distrib. Comput. 71 (2011) 169–184, http://dx.doi.org/10.1016/j.jpdc.2010.08.004.
[60] V.S. Costa, COWL: copy-on-write for logic programs, Poceedings International Parallel Processing Symposium Held Jointly with the Symposium on Parallel and Distributed Processing. (1999) 720–727.
[61] Y. Zhang, Z. Zheng, M.R. Lyu, BFTCloud: a byzantine fault tolerance framework for voluntary-resource cloud computing, Proceedings – IEEE 4th International Conference on Cloud Computing (2011) 444–451, http://dx.doi.org/10.1109/CLOUD.2011.16.
[62] D.A. Menasce, P. Ngo, Understanding cloud computing: experimentation and capacity planning, Proceedings Computer Measurement Group Conf. (2009) 1–11 (10.1.1.158.21).
[63] G. Radhakrishnan, Adaptive application scaling for improving fault-tolerance and availability in the cloud, Bell Labs Tech. J. 17 (2012) 5–14, http://dx.doi.org/10.1002/bltj.21540.
[64] Z. Zheng, T.C. Zhou, M.R. Lyu, I. King, Component ranking for fault-tolerant cloud applications, IEEE Trans. Serv. Comput. 5 (2012) 540–550, http://dx.doi.org/10.1109/TSC.2011.42.
[65] Y.-S. Chen, Y. Pete Chong, Y. Tong, Theoretical foundation of the 80/20 rule, Scientometrics 28 (1993) 183–204, http://dx.doi.org/10.1007/BF02016899.
[66] M. Iqbal, M. Rizwan, Application of 80/20 rule in software engineering waterfall model, 2009 International Conference on Information and Communication Technologies, ICICT 2009 (2009) 223–228, http://dx.doi.org/10.1109/ICICT.2009.5267186.
[67] T.E. Nisonger, The 80/20 rule and core journals, Ser. Libr. 55 (2008) 62–84, http://dx.doi.org/10.1080/03615260801970774.
[68] A. Tyrell, Recovery blocks and algorithm-based fault tolerance, Proceedings 22nd Euromicro Conference (1996) 292–299, http://dx.doi.org/10.1109/EURMIC.1996.546394.
[69] L. Chen, A. Avizienis, N-version programming: a fault-tolerance approach to reliability of software operation, Proceedings Twenty-Fifth International Symposium on Fault-Tolerant Computing (1995) 113–119, http://dx.doi.org/10.1109/FTCSH.1995.532621.
[70] R. Goel, G.M. Shroff, Transparent parallel replication of logically partitioned databases, Proceedings 3rd International Conference on High Performance Computing (1996) 132–137.
[71] S. Yi, A. Andrzejak, D. Kondo, Monetary cost-aware checkpointing and migration on amazon cloud spot instances, IEEE Trans. Serv. Comput. 5 (2012) 512–524.
[72] R. Jhawar, V. Piuri, M. Santambrogio, Fault tolerance management in cloud computing: a system-level perspective, IEEE Syst. J. 7 (2013) 288–297, http://dx.doi.org/10.1109/JSYST.2012.2221934.
[73] J. Cao, M. Simonin, G. Cooperman, C. Morin, Checkpointing as a service in heterogeneous cloud environments, Proceedings – 2015 IEEE/ACM 15th International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2015 (2015) 61–70, http://dx.doi.org/10.1109/CCGrid.2015.160.
[74] D. Poola, K. Ramamohanarao, R. Buyya, Fault-tolerant workflow scheduling using spot instances on clouds, Proceedings International Conference on Computational Science (2014) 523–533, http://dx.doi.org/10.1016/j.procs.2014.05.047 Elsevier Masson SAS.
[75] M. Zhao, F. D'Ugard, K.A. Kwiat, C.A. Kamhoua, Multi-level VM replication based survivability for mission-critical cloud computing, Proceedings 1st International Workshop on Security for Emerging Distributed Network Technologies (2015) 1351–1356.
[76] C.-A. Chen, M. Won, R. Stoleru, G.G. Xie, Energy-efficient fault-tolerant data storage and processing in mobile cloud, IEEE Trans. Cloud Comput. 3 (2015) 28–41, http://dx.doi.org/10.1109/TCC.2014.2326169.
[77] L. Huang, Q. Xu, Lifetime reliability for load-sharing redundant systems with arbitrary failure distributions, IEEE Trans. Reliab. 59 (2010) 319–330, http://dx.doi.org/10.1109/TR.2010.2048679.
[78] L. Al-Awami, H.S. Hassanein, Distributed data storage systems for data survivability in wireless sensor networks using decentralized erasure codes, Comput. Netw. 97 (2016) 113–127, http://dx.doi.org/10.1016/j.comnet.2016.01.008.
[79] J. Wang, W. Bao, X. Zhu, T. Yang, Y. Xiang, FESTAL: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized cloud, IEEE Trans. Comput. 64 (2015) 2545–2558, http://dx.doi.org/10.3969/j.issn.1000-436x.2014.10.020.
[80] J. Ansel, K. Arya, G. Cooperman, DMTCP: transparent checkpointing for cluster computations and the desktop, Proceedings IEEE International Parallel and Distributed Processing Symposium (2009) 1–12, http://dx.doi.org/10.1109/IPDPS.2009.5161063.
[81] Y. Ding, G. Yao, K. Hao, Fault-tolerant elastic scheduling algorithm for workflow in cloud systems, Inf. Sci. (Ny) (2017), http://dx.doi.org/10.1016/j.ins.2017.01.035.
[82] M. Amoon, A framework for providing a hybrid fault tolerance in cloud computing, Proceedings Science and Information Conference (2015) 844–849, http://dx.doi.org/10.1109/SAI.2015.7237242.
[83] M. Amoon, Adaptive framework for reliable cloud computing environment, IEEE Access 4 (2016) 9469–9478, http://dx.doi.org/10.1109/ACCESS.2016.2623633.
[84] Y. Sharma, B. Javadi, W. Si, D. Sun, Reliability and energy efficiency in cloud computing systems: survey and taxonomy, J. Netw. Comput. Appl. 74 (2016) 66–85, http://dx.doi.org/10.1016/j.jnca.2016.08.010.
[85] M.S. Goraya, L. Kaur, Fault tolerance task execution through cooperative computing in grid, Parallel Process. Lett. 23 (2013) 1–20, http://dx.doi.org/10.1142/S0129626413500035.
[86] M. Hasan, M.S. Goraya, A framework for priority based task execution in the distributed computing environment, Proceedings IEEE International Conference on Signal Processing, Computation and Control (2015) 155–158.
[87] M. Amoon, A fault-tolerant scheduling system for computational grids, Comput. Electr. Eng. 38 (2012) 399–412, http://dx.doi.org/10.1016/j.compeleceng.2011.11.004.
[88] A. Chmielowiec, S. Voulgaris, M. Steen, Decentralized group formation, J. Internet Serv. Appl. 5 (2014) 1–18, http://dx.doi.org/10.1186/s13174-014-0012-2.
[89] S. Bansal, P. Kumar, K. Singh, Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs, J. Parallel Distrib. Comput. 65 (2005) 479–491, http://dx.doi.org/10.1016/j.jpdc.2004.11.006.
[90] S. Ren, M. van der Schaar, Dynamic scheduling and pricing in wireless cloud computing, IEEE Trans. Mob. Comput. 13 (2014) 2283–2292, http://dx.doi.org/10.1109/TMC.2013.57.
[91] R. Birke, I. Giurgiu, L.Y. Chen, D. Wiesmann, T. Engbersen, Failure analysis of virtual and physical machines: patterns, causes and characteristics, Proc. Int. Conf.

Depend. Syst. Netw. (2014) 1–12, http://dx.doi.org/10.1109/DSN.2014.18.
[92] S.M. Abdulhamid, M.S. Abd Latiff, S.H.H. Madni, M. Abdullahi, Fault tolerance aware scheduling technique for cloud computing environment using dynamic clustering algorithm, Neural Comput. Appl. (2016) 1–15, http://dx.doi.org/10.1007/s00521-016-2448-8.
[93] W. Chen, R.F. Da Silva, E. Deelman, T. Fahringer, Dynamic and fault-tolerant clustering for scientific workflows, IEEE Trans. Cloud Comput. 4 (2016) 49–62, http://dx.doi.org/10.1109/TCC.2015.2427200.
[94] H. Idris, A.E. Ezugwu, S.B. Junaidu, A.O. Adewumi, An improved ant colony optimization algorithm with fault tolerance for job scheduling in grid computing systems, PLoS One 12 (2017) 1–24, http://dx.doi.org/10.1371/journal.pone.0177567.
[95] A. Moghtadaeipour, R. Tavoli, A new approach to improve load balancing for increasing fault tolerance and decreasing energy consumption in cloud computing, International Conference on Knowledge-Based Engineering and Innovation (2015) 982–987.
[96] A. Thakur, M.S. Goraya, A taxonomic survey on load balancing in cloud, J. Netw. Comput. Appl. 98 (2017) 43–57, http://dx.doi.org/10.1016/j.jnca.2017.08.020.
[97] N. Garg, M.S. Goraya, Task deadline-aware energy-efficient scheduling model for a virtualized cloud, Arab. J. Sci. Eng. (2017), http://dx.doi.org/10.1007/s13369-017-2779-5.

Moin Hasan received the B.E. degree in computer science and engineering from Sant Longowal Institute of Engineering and Technology, Sangrur, India, in 2012 and the master's degree in computer science and engineering from Mangalayatan University, Aligarh, India, in 2014. He then worked as a Research Scholar in the Department of Computer Science and Engineering, Sant Longowal Institute of Engineering and Technology, Sangrur, India, until March 2018. His areas of research interest include scheduling, load balancing and fault tolerance in parallel and distributed systems, including grid and cloud computing.

Major S. Goraya received the B.E. degree in computer science and engineering from Sant Longowal Institute of Engineering and Technology, Sangrur, India, in 1997 and the master's and PhD degrees in computer science and engineering from Punjabi University, Patiala, India, in 2003 and 2013, respectively. He is currently working as Associate Professor in the Department of Computer Science and Engineering, Sant Longowal Institute of Engineering and Technology, Sangrur, India. His research interests include grid computing, cloud computing, and distributed computing.