CHAPTER

15

A TAXONOMY AND SURVEY OF FAULT-TOLERANT WORKFLOW MANAGEMENT SYSTEMS IN CLOUD AND DISTRIBUTED COMPUTING ENVIRONMENTS

Deepak Poola[*], **Mohsen Amini Salehi**[†], **Kotagiri Ramamohanarao**[*], **Rajkumar Buyya**[*]
[*]*The University of Melbourne, Australia* [†]*The University of Louisiana Lafayette, USA*

15.1 INTRODUCTION

Workflows orchestrate the relationships between dataflow and computational components by managing their inputs and outputs. In the recent years, scientific workflows have emerged as a paradigm for managing complex large scale distributed data analysis and scientific computation. Workflows automate computation, and thereby accelerate the pace of scientific progress easing the process for researchers. In addition to automation, it is also extensively used for scientific reproducibility, result sharing and scientific collaboration among different individuals or organizations. Scientific workflows are deployed in diverse distributed environments, starting from supercomputers and clusters, to grids and currently cloud computing environments [1,2].

Distributed environments usually are large scale infrastructures that accelerate complex workflow computation; they also assist in scaling and parallel execution of the workflow components. The likelihood of failure increases specially for long-running workflows [3]. However, these environments are prone to performance variations and different types of failures. This demands the workflow management systems to be robust against performance variations and fault-tolerant against faults.

Over the years, many different techniques have evolved to make workflow scheduling fault-tolerant in different computing environments. This chapter aims to categorize and classify different fault-tolerant techniques and provide a broad view of fault-tolerance in workflow domain for distributed environments.

Workflow scheduling is a well studied research area. Yu et al. [4] provided a comprehensive view of workflows, different scheduling approaches, and different workflow management systems. However, this work did not throw much light into fault-tolerant techniques in workflows. Plankensteiner et al. [5] have recently studied different fault-tolerant techniques for grid workflows. Nonetheless, they do not provide a detailed view into different fault-tolerant strategies and their variants. More importantly, their work does not encompass other environments like clusters and clouds.

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00015-6

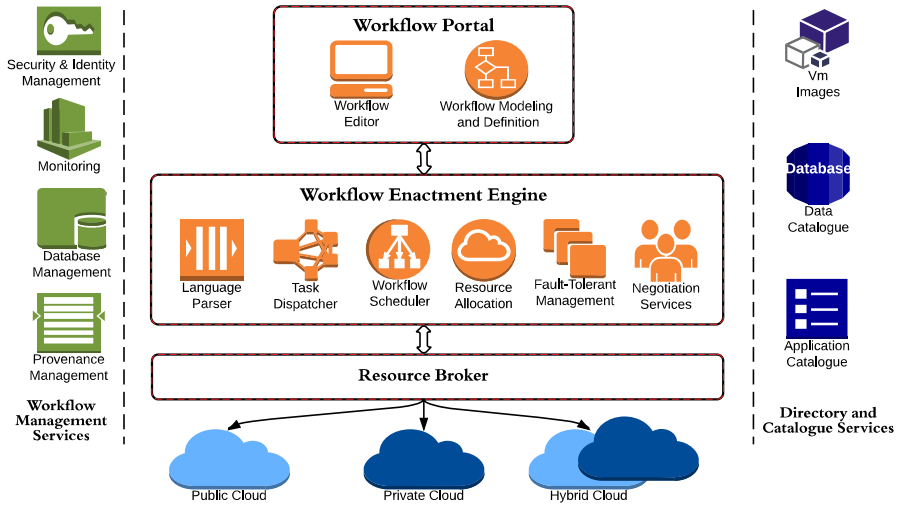

FIGURE 15.1

Architecture of cloud workflow management system. Portal, enactment engine, and resource broker form the core of the WFMS performing vital operations, such as designing, modeling, and resource allocation. To achieve these operations, the workflow management services (left column) provide security, monitoring, database, and provenance management services. In addition, the Directory and Catalogue services (right column) provide catalog and metadata management for the workflow execution

In this chapter, we aim to provide a comprehensive taxonomy of fault-tolerant workflow scheduling techniques in different existing distributed environments. We first start with an introduction to workflows and workflow scheduling. Then, we introduce fault-tolerance and its necessity. We provide an in-depth ontology of faults in Section 15.4. Following which, different fault-tolerant workflow techniques are detailed. In Section 15.6, we describe different approaches used to model failures and also give definition of various metrics used in literature to assess fault-tolerance. Finally, prominent workflow management systems are introduced and a description of relevant tools and support systems that are available for workflow development is provided.

15.2 BACKGROUND

15.2.1 WORKFLOW MANAGEMENT SYSTEMS

Workflow management systems (WFMS) enable automated and seamless execution of workflows. It allows users to define and model workflows, set their deadline and budget limitations, and the environments in which they wish to execute. The WFMS then evaluates these inputs and executes them within the defined constraints.

The prominent components of a typical cloud WFMS is given in Fig. 15.1. The *workflow portal* is used to model and define abstract workflows, i.e., tasks and their dependencies. The *workflow en-*

actment engine takes the abstract workflows and parses them using a language parser. Then, the *task dispatcher* analyzes the dependencies and dispatches the ready tasks to the scheduler. The *scheduler*, based on the defined scheduling algorithms, schedules the workflow task onto a resource. We further discuss about workflow scheduling in the next section. Workflow enactment engine also handles the fault-tolerance of the workflow. It also contains a resource allocation component which allocates resources to the tasks through the resource broker.

The *resource broker* interfaces with the infrastructure layer and provides a unified view to the enactment engine. The resource broker communicates with compute services to provide the desired resource.

The *directory and catalogue services* house information about data objects, the application and the compute resources. This information is used by the enactment engine, and the resource broker to make critical decisions.

Workflow management services, in general, provide important services that are essential for the working of a WFMS. *Security and identify* services ensure authentication and secure access to the WFMS. *Monitoring* tools constantly monitor vital components of the WFMS and raise alarms at appropriate times. *Database management* component provides a reliable storage for intermediate and final data results of the workflows. *Provenance management services* capture important information such as, dynamics of control flows and data, their progressions, execution information, file locations, input and output information, workflow structure, form, workflow evolution, and system information [6]. Provenance is essential for interpreting data, determining its quality and ownership, providing reproducible results, optimizing efficiency, troubleshooting and also to provide fault-tolerance.

15.2.2 WORKFLOW SCHEDULING

As mentioned earlier, a workflow is a collection of tasks connected by control and/or data dependencies. Workflow structure indicates the temporal relationship between tasks. Workflows can be represented either in Directed Acyclic Graph (DAG) (as shown in Fig. 15.3) or non-DAG formats.

Scheduling in workflows maps its tasks on to distributed resources such that the dependencies are not violated. Workflow Scheduling is a well-known NP-Complete problem [7].

The workflow scheduling architecture specifies the placement of the scheduler in a WFMS and it can be broadly categorized into three types as illustrated in Fig. 15.2: *centralized*, *hierarchical*, and *decentralized* [4]. In the *centralized* approach, a centralized scheduler makes all the scheduling decisions for the entire workflow. The drawback of this approach is that it is not scalable; however, it can produce efficient schedules as the centralized scheduler has all the necessary information. In *hierarchical* scheduling, there is a central manager responsible for controlling the workflow execution and assigning the subworkflows to low-level schedulers. The low-level schedulers map tasks of the subworkflows assigned by the central manager. In contrast, *decentralized* scheduling has no central controller. It allows tasks to be scheduled by multiple schedulers, each scheduler communicates with each other and schedules a subworkflow or a task [4].

Workflow schedule planning for workflow applications also known as planning scheme are of two types: *static(offline)* and *dynamic(online)*. *Static* scheme map tasks to resources at the compile time. These algorithms require the knowledge of workflow tasks and resource characteristics beforehand. On the contrary, *dynamic* scheme can make few assumptions before execution and make scheduling

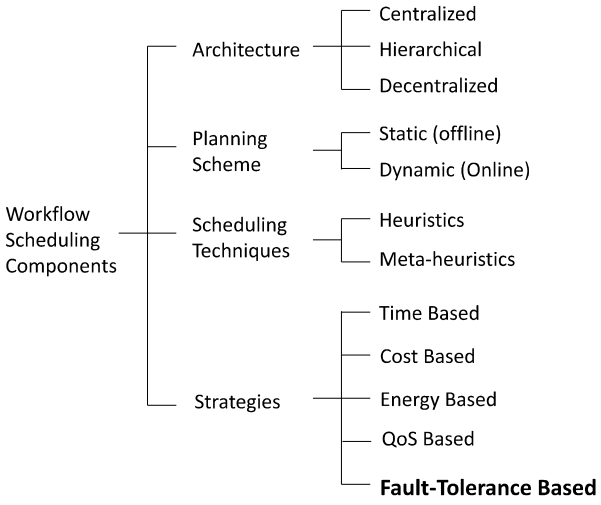

FIGURE 15.2

Components of workflow scheduling

decision just-in-time [8]. Here, both dynamic and static information about environment is used in scheduling decisions.

Further, workflow scheduling techniques are the approaches or methodologies used to map workflow tasks to resources, and it can be classified into two types: *heuristics* and *metaheuristics*. *Heuristic* solutions exploit problem-dependent information to provide an approximate solution trading optimality, completeness, accuracy, and/or processing speed. It is generally used when finding a solution through exhaustive search is impractical. It can be further classified into list based scheduling, cluster based scheduling, and duplication based algorithms [9,10]. On the other hand, *metaheuristics* are more abstract procedures that can be applied to a variety of problems. A metaheuristic approach is problem-independent and treats problems like black boxes. Some of the prominent metaheuristic approaches are genetic algorithms, particle swarm optimization, simulated annealing, and ant colony optimization.

Each scheduling algorithm for any workflow has one or many objectives. The most prominent strategies or objectives used are given in Fig. 15.2. Time, cost, energy, QoS, and fault-tolerance are most commonly used objectives for a workflow scheduling algorithm. Algorithms can be with a single objective or multiple objectives based on the scenario and the problem statement. The rest of the chapter is focused on scheduling algorithms and workflow management systems whose objective is fault-tolerance.

15.3 INTRODUCTION TO FAULT-TOLERANCE

Failure is defined as any deviation of a component of the system from its intended functionality. Resource failures are not the only reason for the system to be unpredictable, factors such as, design faults,

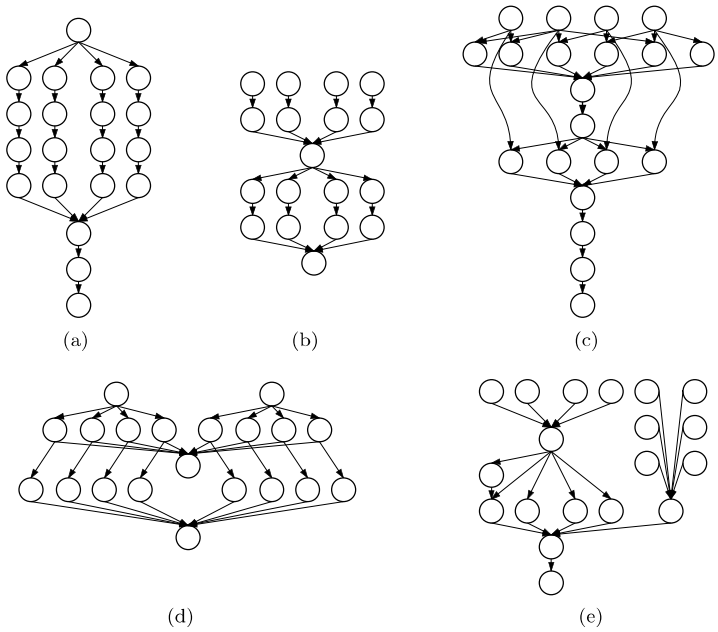

FIGURE 15.3

Examples of the state-of-the-art workflows [11]: (a) Epigenomics: DNA sequence data obtained from the genetic analysis process is split into several chunks and are used to map the epigenetic state of human cells. (b) LIGO: detects gravitational waves of cosmic origin by observing stars and black holes. (c) Montage: creates a mosaic of the sky from several input images. (d) CyberShake: uses the Probabilistic Seismic Hazard Analysis (PSHA) technique to characterize earthquake hazards in a region. (e) SIPHT: searches for small untranslated RNAs encoding genes for all of the bacterial replicas in the NCBI database

performance variations in resources, unavailable files, and data staging issues can be few of the many reasons for unpredictable behaviors.

Developing systems that tolerate these unpredictable behaviors and provide users with seamless experience is the aim of fault-tolerant systems. Fault tolerance is to provide correct and continuous operation albeit faulty components. Fault-tolerance, robustness, reliability, resilience and Quality of Service (QoS) are some of the ambiguous terms used for this. These terminologies are used interchangeably in many works. Significant works are done in this area encompassing numerous fields like job-shop scheduling [12], supply chain [13], and distributed systems [10,14].

Any fault-tolerant WFMS need to address three important questions [14]: (a) What are the factors or uncertainties that the system is fault-tolerant towards? (b) What behavior makes the system fault-tolerant? (c) How to quantify the fault-tolerance, i.e., what is the metric used to measure fault-tolerance?

In this survey we categorize and define the taxonomy of various types of faults that a WFMS in a distributed environment can experience. We further develop an ontology of different fault-tolerant

mechanisms that are used until now. Finally we provide numerous metrics that measure fault-tolerance of a particular scheduling algorithm.

15.3.1 NECESSITY FOR FAULT-TOLERANCE IN DISTRIBUTED SYSTEMS

Workflows, generally, are composed of thousands of tasks, with complicated dependencies between the tasks. For example, some prominent workflows (as shown in Fig. 15.3) widely considered are Montage, CyberShake, Broadband, Epigenomics, LIGO Inspiral Analysis, and SIPHT, which are complex scientific workflows from different domains such as astronomy, life sciences, physics, and biology. These workflows are composed of thousands of tasks with various execution times, which are interdependent.

The workflow tasks are executed on distributed resources that are heterogeneous in nature. WFMSs that allocates these workflows uses middleware tools that require to operate congenially in a distributed environment. This very complex and complicated nature of WFMSs and its environment invite numerous uncertainties and chances of failures at various levels.

In particular, in data-intensive workflows that continuously process data, machine failure is inevitable. Thus, failure is a major concern during the execution of data-intensive workflows frameworks, such as MapReduce and Dryad [15]. Both transient (i.e., fail-recovery) and permanent (i.e., fail-stop) failures can occur in data-intensive workflows [16]. For instance, Google reported on average 5 permanent failures in form of machine crashes per MapReduce workflow during March 2006 [17] and at least one disk failure in every run of MapReduce workflow with 4000 tasks.

Necessity for fault-tolerance arises from this very nature of the application and environment. Workflows are applications that are most often used in a collaborative environment spread across the geography involving various people from different domains (e.g., [11]). So many diversities are potential causes for adversities. Hence, to provide a seamless experience over a distributed environment for multiple users of a complex application, fault-tolerance is a paramount requirement of any WFMS.

15.4 TAXONOMY OF FAULTS

Fault is defined as a defect at the lowest level of abstraction. A change in a system state due to a fault is termed as an error. An error can lead to a failure, which is a deviation of the system from its specified behavior [18,19]. Before we discuss about fault-tolerant strategies it is important to understand the fault-detection and identification methodologies and the taxonomy of faults.

Faults can be characterized in an environment through various elements and means. Lackovic et al. [20] provide a detailed list of these element that are illustrated in Fig. 15.4. *Accuracy* of fault detection can be either known or unknown faults. Known faults are those which have been reported before and solutions for such faults are known. *Location* is the part of the environment where the fault occurs. *Originator* is the part of the environment responsible for the fault to occur. *Stage* of the fault refers to the phase of the workflow lifecycle (design, build, testing, and production) when the fault occurred. *Time* is the incidence time in the execution when the fault happened. *Frequency*, as the name suggests identifies the frequency of fault occurrence. *Severity* specifies the difficulty in taking the corrective measures and details the impact of a particular fault. More details of these elements can be found in [20].

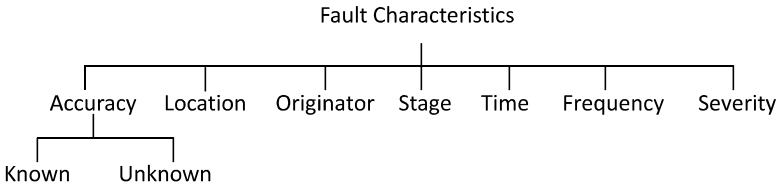

FIGURE 15.4

Elements through which faults can be characterized

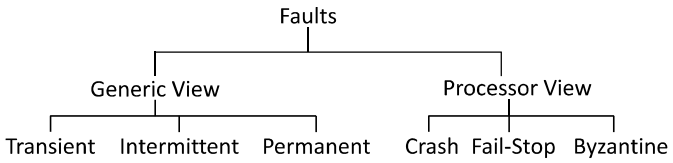

FIGURE 15.5

Faults: views and their classifications

At a high level, faults can be viewed in two different ways, *generic view*, and the *processor view*. The *generic view* of faults can be classified into three major types as shown in Fig. 15.5: *transient*, *intermittent* and *permanent* [20]. *Transient* faults invalidate only the current task execution, on a rerun or restart these fault most likely will not manifest again [21]. *Intermittent* faults appear at infrequent intervals. Finally, *permanent* faults are faults whose defects cannot be reversed.

From a *processor's perspective*, faults can be classified into three classes: *crash*, *fail-stop*, and *byzantine* [19]. This is mostly used for resource or machine failures. In the *crash* failure model, the processor stops executing suddenly at a specific point. In *fail-stop* processor's internal state is assumed to be volatile. The contents are lost when a failure occurs and it cannot be recovered. However, this class of failure does not perform an erroneous state change due to a failure [22]. *Byzantine* faults originate due to random malfunctions like aging or external damage to the infrastructure. These faults can be traced to any processor or messages [23].

Faults in a workflow environment can occur at different levels of abstraction [5]: hardware, operating system, middleware, task, workflow, and user. Some of the prominent faults that occur are network failures, machine crashes, out-of-memory, file not found, authentication issues, file staging errors, uncaught exceptions, data movement issues, and user-defined exceptions. Plankensteiner et al. [5] detail various faults and map them to different level of abstractions.

15.5 TAXONOMY OF FAULT-TOLERANT SCHEDULING ALGORITHMS

This section details the workings of various fault-tolerant techniques used in WFMS. In the rest of this section, each technique is analyzed and their respective taxonomies are provided. Additionally,

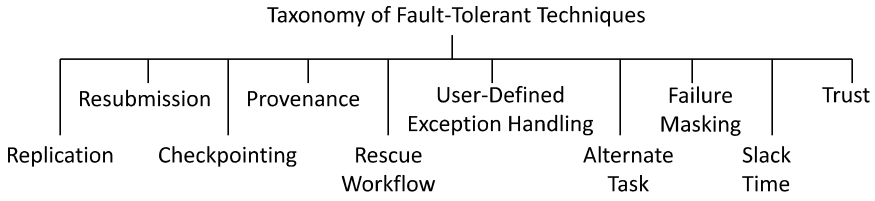

FIGURE 15.6

Taxonomy of workflow scheduling techniques to provide fault-tolerance

prominent works using each of these techniques are explained. Fig. 15.6 provides an overview of various techniques that are used to provide fault-tolerance.

15.5.1 REPLICATION

Redundancy in space is one of the widely used mechanisms for providing fault-tolerance. Redundancy in space means providing additional resources to execute the same task to provide resilience and it is achieved by duplication or replication of resources. There are broadly two variants of redundancy of space, namely, task duplication and data replication.

15.5.1.1 Task duplication

Task duplication creates replica of tasks. Replication of tasks can be done concurrently [24], where all the replicas of a particular task start executing simultaneously. When tasks are replicated concurrently, the child tasks start its execution depending on the schedule type. Fig. 15.7 illustrates the taxonomy of task duplication.

Schedules types, are either *strict* or *lenient*. In *strict* schedule the child task executes only when all the replicas have finished execution [25]. In the *lenient* schedule type, the child tasks start execution as soon as one of the replicas finishes execution [24].

Replication of task can also be performed in a backup mode, where the replicated task is activated when the primary tasks fail [26]. This technique is similar to retry or redundancy in time. However, here, they employ a backup overloading technique, which schedules the backups for multiple tasks in the same time period to effectively utilize the processor time.

Duplication is employed to achieve multiple objectives, the most common being fault-tolerance [25, 27–29]. When one task fails, the redundant task helps in completion of the execution. Additionally, algorithms employ data duplication where data is replicated and prestaged, thereby moving data near computation especially in data intensive workflows to improve performance and reliability [30]. Furthermore, estimating task execution time a priori in a distributed environment is arduous. Replicas are used to circumvent this issue using the result of the earliest completed replica. This minimizes the schedule length to achieve hard deadlines [31–34], as it is effective in handling performance variations [24]. Calheiros et al. [35] replicated tasks in idle time slots to reduce the schedule length. These replicas also increase resource utilization without any extra cost.

Task duplication is achieved by replicating tasks in either *idle cycles* of the resources or *exclusively on new resources*. Some schedules use a *hybrid approach* replicating tasks in both idle cycles and

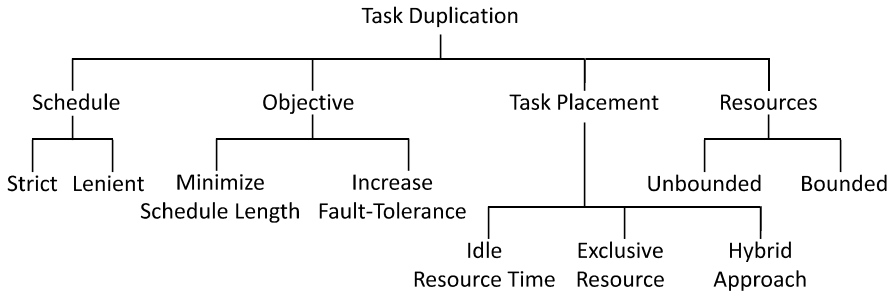

FIGURE 15.7

Different aspects of task duplication technique in providing fault-tolerance

new resources [35]. Idle cycles are those time slots in the resource usage period where the resources are unused by the application. Schedules that replicate in these idle cycles profile resources to find unused time slot, and replicate tasks in those slots. This approach achieves benefits of task duplication and simultaneously considers monetary costs. In most cases, however, these idle slots might not be sufficient to achieve the needed objective. Hence, task duplication algorithms commonly place their task replicas on new resources. These algorithms trade off resource costs to their objectives.

There is a significant body of work in this area encompassing platforms like cluster, grids, and clouds [31,25,27–29,32–34,36]. Resources considered can either be *bounded* or *unbounded* depending on the platform and the technique. Algorithms with bounded resources consider a limited set of resources. Similarly, an unlimited number of resources are assumed in an unbounded system environment. Resource types used can either be *homogeneous* or *heterogeneous* in nature. *Homogeneous* resources have similar characteristics, and *heterogeneous* resources on the contrary vary in their characteristics such as, processing speed, CPU cores, memory, etc. Darbha et al. [31] is one of the early works, which presents an enhanced search and duplication based scheduling algorithm (SDBS) that takes into account the variable task execution time. They consider a distributed system with homogeneous resources and assume an unbounded number of processors in their system.

15.5.1.2 Data replication

Data in workflows are either not replicated (and are stored locally by the processing machines) or is stored on the distributed file system (DFS) where it is automatically replicated (e.g., in Hadoop Distributed File System (HDFS)). Although the former approach is efficient, particularly in data-intensive workflows, it is not fault-tolerant. That is, failure of a server storing data causes the re-execution of the affected tasks. On the other hand, the latter approach offers more fault tolerance but is not efficient due to significant network overhead and increasing the execution time of the workflow.

Hadoop, which is a platform for executing data-intensive workflows, uses a static replication strategy for fault-tolerance. That is, users can manually determine the number of replicas that have to be created from the data. Such static and blind replication approach imposes a significant storage overhead to the underlying system (e.g., cluster or cloud) and slows down the execution of the MapReduce workflow. One approach to cope with this problem is to adjust the replication rate dynamically based on the usage rate of the data. This will reduce the storage and processing cost of the resources [37].

Cost-effective incremental replication (CIR) [38] is a strategy for cloud based workflows that predicts when a workflow is needed to replicate to ensure the reliability requirement of the workflow execution.

There are four major data-replication methods for data-intensive workflows on large-scale distributed systems (e.g., clouds) namely, *synchronous* and *asynchronous* replication, *rack-level* replication, and *selective* replication. These replication methods can be applied on input, intermediate, or output data of a workflow.

In *synchronous* data replication, such as those in HDFS, writers (i.e., producer tasks in a workflow) are blocked until replication finishes. Synchronous replication method leads to a high consistency because if a writer of block A returns, all the replicas of block A are guaranteed to be identical and any reader (i.e., consumer tasks in a workflow) of block A can read any replica. Nonetheless, the drawback of this approach is that the performance of writers might get affected as they have to be blocked. In contrast, *asynchronous* data replication [16] allows writers to proceed without waiting for a replication to complete. The asynchronous data replication consistency is not as accurate as the synchronous method because even if a writer of block A returns, a replica of block A may still be in the replication process. Nonetheless, performance of the writers improves due to the nonblocking nature. For instance, with an asynchronous replication in Hadoop, Map and Reduce tasks can proceed without being blocked.

Rack-level data replication method enforces replication of the data blocks on the same rack in a data center. In cloud data centers machines are organized in racks with a hierarchical network topology. A two-level architecture with a switch for each rack and a core switch is a common network architecture in these data centers. In this network topology the core switch can become bottleneck as it is shared by many racks and machines. That is, there is heterogeneity in network bandwidth where inter-rack bandwidth is scarce compared to intra-rack bandwidth. One example of bandwidth bottleneck is in the Shuffling phase of MapReduce. In this case, as the communication pattern between machines is all-to-all, the core switches become over-utilized whereas rack-level switches are underutilized. Rack-level replication reduces the traffic transferred through the bandwidth-scarce core switch. However, the drawback of the rack-level replication approach is that it cannot tolerate rack-level failures and if a rack fails, all the replicas become unavailable. There are observations that show rack-level failures are infrequent which proves the efficacy of rack-level replication. For instance, one study shows that Google experiences approximately 20 rack failures within a year [39].

Selective data replication is an approach where the data generated by the previous step of the workflow are replicated on the same machine, where they are generated. For instance, in a chained MapReduce workflow, once there is a machine failure at the Map phase, the affected Map tasks can be restarted instantly, if the data generated by the previous Reduce tasks were replicated locally on the same machine. In this manner, the amount of intermediate data that needs to be replicated in the Map phase is reduced remarkably. However, it is not very effective for Reduce phase, because Reduce data are mostly locally consumed.

ISS [16] is a system that extends the APIs of HDFS and implements a combination of three aforementioned replication approaches. It implements a rack-level replication that asynchronously replicates locally-consumed data. The focus of ISS is on the management of intermediate data in Hadoop data-intensive workflows. It takes care of all aspects of managing intermediate data such as writing, reading, Shuffling, and replicating. Therefore, a programming framework that utilizes ISS does not need to consider Shuffling. ISS transparently transfers intermediate data from writers (e.g., Map tasks) to readers (e.g., Reduce tasks).

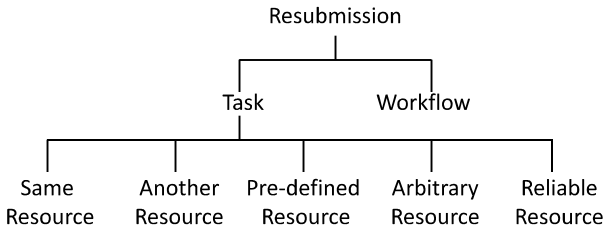

FIGURE 15.8

Taxonomy of resubmission fault-tolerant technique

As mentioned earlier, replicating input data or intermediate data on stable external storage systems (e.g., distributed file systems) is expensive for data-intensive workflows. The overhead is due to data replication, disk I/O, network bandwidth, and serialization which can potentially dominate the workflow execution time [40]. To avoid these overheads, in frameworks such as Pregel [41], which is a system for iterative graph computation, intermediate data are maintained in memory. Resilient Distributed Datasets (RDDs) [40] enables data reuse in a fault-tolerant manner. RDDs are parallel data structures that enable users to persist intermediate data in memory and manipulate them using various operators. It also controls the partitioning of the data to optimize data placement. RDD has been implemented within the Spark [42] framework.

15.5.2 RESUBMISSION

Resubmission tries to reexecute components to mitigate failures. Resubmission or redundancy in time helps recover from transient faults or soft errors. Resubmission is employed as an effective fault-tolerant mechanism by around 80% of the WFMSs [5]. Li et al. [43] claim that 41% of failures are recovered in their work through resubmission. Some of the WFMS that support resubmission for fault-tolerance are Askalon, Chemomentum, GWES, Pegasus, P-Grade, Proactive, Triana, and Unicore [5].

Resubmission can be classified into two levels: *workflow* and *task* resubmission as illustrated in Fig. 15.8. In workflow resubmission, as the name suggests, the entire application or a partial workflow is resubmitted [44].

Task resubmission, retries the same task to mitigate failure. Task retry/resubmission can be either done on the same resource or another resource [5]. Resubmission on the same resource is applicable when a task fails due to a transient failure or due to file staging issues. In other cases this might not be the best approach to mitigate failures. Resubmission of the task can be either done on a fixed predefined resource [45] or on an arbitrary resource or a resource with high reliability. A fixed predefined resource is not necessarily the same resource, but the drawbacks are similar to that. Selecting a resource arbitrarily without a strategy is not the most effective solution to avoid failures. Employing a strategy whilst selecting resources, like choosing resources with high reliability, increases the probability of addressing failures. Zhang et al. [28] rank resources based on a metric called reliability prediction and use this metric to schedule their task retries.

Resources considered can either be homogeneous or heterogeneous in nature. In a heterogeneous resource type environment, different resource selection strategies have different impact on cost and

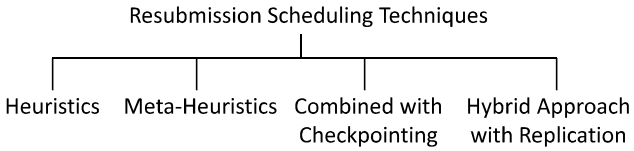

FIGURE 15.9

Different approaches used in resubmission algorithms

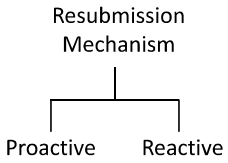

FIGURE 15.10

Classification of resubmission mechanisms

time. A dynamic algorithm must take into consideration deadline and budget restrictions, and select resources that provide fault-tolerance based on these constraints. Clouds providers like Amazon, offer resources in an auction-like mechanism for low cost with low SLAs called spot instances. Poola et al. [46] have proposed a just-in-time dynamic algorithm that uses these low cost instances to provide fault-tolerant schedules considering the deadline constraint. They resubmit tasks upon failures to either spot or on-demand instances based on the criticality of the workflow deadline. This algorithm is shown to provide fault-tolerant schedule whilst reducing costs.

Algorithms usually have a predefined limit for the number of retries that they will attempt [28,47] to resolve a failure. Some algorithms also have a time interval in addition to the number of retries threshold [45]. However, there are algorithms that consider infinite retries as they assume the faults to be transient in nature [48].

Algorithms using resubmission can be broadly classified into four types as shown in Fig. 15.9: *Heuristic* based [43,45,49], *metaheuristic* based [44], *hybrid of resubmission and checkpointing* [28], and *hybrid of resubmission and replication* [50]. Heuristic based approaches are proven to be highly effective, although these solutions are specific to a particular use case and take lot of assumptions. Metaheuristics provide near optimal solutions and are more generic approaches; however, they are usually time and memory consuming. Employing hybrid approaches with checkpointing saves time, does not perform redundant computing, and does not overutilize resources. However, these approaches delay the makespan as resubmission retries a task in case of failures, although, checkpointing reruns from a saved state it still requires additional time delaying the makespan. Replication with redundant approaches wastes resources but does not delay the makespan as the replicas eliminate the necessity of rerunning a task.

Finally, resubmission fault-tolerant mechanism is employed in two major ways (Fig. 15.10): *proactive* and *reactive*. In the *proactive* mechanism [44,51], the algorithm predicts a failure or a performance

slowdown of a machine and reschedules it on another resource to avoid delays or failures. In *reactive* mechanism, the algorithms resubmit tasks or a workflow after a failure occurs.

Resubmission in workflow provides resilience for various faults. However, the drawback of this mechanism is the degradation in the total execution time when large number of failures occurs. Resubmission is ideal for an application during the execution phase and replication is well suited at the scheduling phase [50].

15.5.3 CHECKPOINTING

Checkpointing is an effective and widely used fault-tolerant mechanism. In this process, states of the running process are periodically saved to a reliable storage. These saved states are called *checkpoints*. Checkpointing restores the saved state after a failure, i.e., the process will be restarted from its last checkpoint or the saved state. Depending on the host, we can restart the process on the same machine (if it has not failed) or on another machine [52,23]. WFMS actively employ checkpointing as their fault-tolerant mechanism. More than 60% of these systems use checkpointing to provide resilience [5].

A checkpoint data file typically contains data, states and stack segments of the process. It also stores information of open files, pending signals and CPU states [53].

15.5.3.1 Checkpoint selection strategies

How often or when to take checkpoints is an important question while checkpointing. Various systems employ different checkpoint selection strategies. Prominent selection strategies are [53–56]:

- Using event activity as a checkpoint;
- Taking checkpoints at the start and end time of an activity;
- Taking a checkpoint at the beginning and then after each decision activity;
- Letting user define some static stages during build-stage;
- Taking checkpoint when runtime completion duration is greater than maximum activity duration;
- Taking checkpoint when runtime completion duration is greater than mean duration of the activity;
- Reacting when an activity fails;
- Reacting when an important activity finishes completion;
- Reacting after a user defined deadline (e.g., percentage of workflow completion);
- Reacting to system changes like availability of services.
- Using application defined stages;
- Reacting based on linguistic constructs for intervention of programmers.

15.5.3.2 Issues and challenges

Checkpointing provides fault-tolerance against transient faults only. If there is a design fault, checkpointing cannot help recover from it [57]. Another challenge here is to decide on the number of checkpoints to be taken. The more the checkpoints, the higher the overhead, whereas fewer checkpoints leads to excessive loss of computation [58]. The overhead imposed by checkpointing depends on the level that it is applied (e.g., process or virtual machine level). A mathematical model is provided in [59] to calculate the checkpointing overhead of virtual machines.

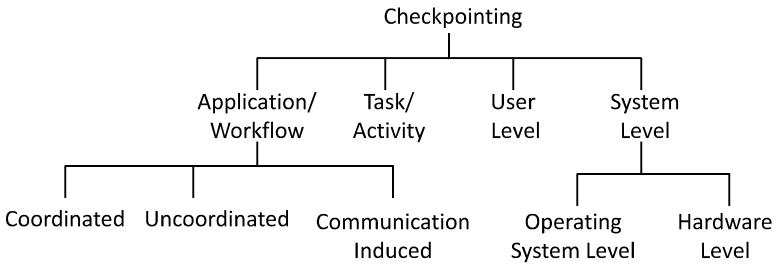

FIGURE 15.11

Taxonomy of checkpointing mechanism

In message-passing systems inter-process dependencies are introduced by messages. When one or more processes fail, these dependencies may lead to a restart even if the processes did not fail. This is called rollback propagation that may lead the system to the initial state. This situation is called a domino effect [54]. The domino effect occurs if checkpoints are taken independently in an uncoordinated fashion in a system. This can be avoided by performing checkpoints in a coordinated manner. Further, if checkpoints are taken to maintain system-wide consistency then the domino effect can be avoided [54].

15.5.3.3 Taxonomy of checkpointing

As shown in Fig. 15.11, there are four major checkpointing approaches: *Application/workflow-level*, *task/activity level*, *user level*, and *system level* implementation.

In *application/workflow-level* checkpointing, implementation is usually performed within the source code, or is automatically injected into the code using external tools. It captures the state of the entire workflow and its intermediate data [58,53]. This can be further classified into *coordinated*, *uncoordinated*, or *communication-induced* [54]. Coordinated approach takes checkpoints in a synchronized fashion to maintain a global state. Recovery in this approach is simple and the domino effect is not experienced in this method. It maintains only one permanent checkpoint on a reliable storage, eliminating the need for garbage collection. The drawback is incurring a large latency in committing the output [54].

Coordinated checkpointing can further be achieved in the following ways: *Nonblocking* Checkpoint Coordination, Checkpointing with *Synchronized Clocks*, *Checkpointing and Communication Reliability*, and *Minimal Checkpoint Coordination*.

Nonblocking Checkpoint Coordination: Here, the initiator takes a checkpoint and broadcasts a checkpoint request to all other activities. Each activity or task takes a checkpoint once it receives this checkpoint request and then further rebroadcasts the request to all tasks/activities.

Checkpointing with Synchronized Clocks: This approach is done with loosely synchronized clocks that trigger local checkpointing for all activities without an initiator.

Checkpointing and Communication Reliability: This protocol saves all the in-transit messages by their destination tasks. These messages are not saved when reliable communication channels are not assumed.

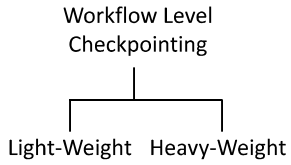

FIGURE 15.12

Workflow-level checkpointing

Minimal Checkpoint Coordination: In this case, only a minimum subset of the tasks/activities is saved as checkpoints. The initiator identifies all activities with which it has communicated since the last checkpoint and sends them a request. Upon receiving the request, each activity further identifies other activities it has communicated since the last checkpoint and sends them a request.

Uncoordinated checkpointing allows each task to decide the frequency and time to saved states. In this method, there is a possibility of domino effect. As this approach is not synchronized, it may take many useless checkpoints that are not part of the global consistent state. This increases overhead and do not enhance the recovery process. Multiple uncoordinated checkpoints force garbage collection to be invoked periodically.

The last type of workflow-level checkpointing is *Communication-Induced Checkpointing*. In this protocol the information about checkpointing is piggybacked in the application messages. The receiver then uses this information to decide whether or not to checkpoint.

Based on the intermediate data, workflow-level checkpointing can also be subcategorized into two types: *lightweight* and *heavyweight* as illustrated in Fig. 15.12. In *lightweight* checkpointing the intermediate data is not stored, only a reference to it is stored assuming that the storage is reliable. Alternatively, *heavyweight* checkpointing stores the intermediate data along with other things in a checkpoint [58,53].

Task-level checkpointing saves the register, stack, memory, and intermediate states for every individual task running on a virtual machine [60] or a processor [58,53]. When a failure occurs, the task can restart from the intermediate saved state and this is especially important when the failures are independent. This helps recover individual units of the application.

User-level checkpointing uses a library to do checkpoints and the application programs are linked to it. This mechanism is not transparent as the applications are modified, recompiled and relinked. The drawback being this approach cannot checkpoint certain shell scripts, system calls, and parallel application as the library may not be able access system files [57].

System-level checkpointing can be done either at the *operating system level* or the *hardware level*. This mechanism is transparent to the user and it does not necessarily modify the application program code. The problem with operating system level checkpointing is that it cannot be portable and modification at the kernel level is not always possible and difficult to achieve [57].

15.5.3.4 Performance optimization

As discussed earlier, optimizing performance in a checkpoint operation is a challenge. The frequency of checkpoints impacts the storage and computation load. Checkpointing schemes can be broadly divided into *online* and *offline* checkpointing schemes as illustrated in Fig. 15.13.

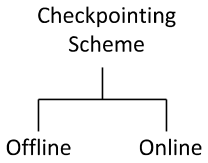

FIGURE 15.13

Checkpointing schemes

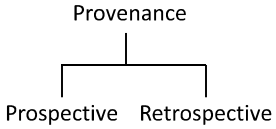

FIGURE 15.14

Forms of provenance

An *offline* checkpointing scheme determines the frequency for a task before its execution. The drawback being it is not an adaptive approach. On the other hand, *online* schemes determine the checkpointing interval dynamically based on the frequency of fault occurrences and the workflow deadline. The dynamic checkpointing is more adaptive and is able to optimize performance of the WFMS.

15.5.3.5 Checkpointing in WFMS

WFMSs employ checkpointing at various levels. At Workflow-level, two types of checkpointing can be employed lightweight and heavyweight as stated earlier. Lightweight checkpointing is used by Chemomentum, GWEE, GWES, Pegasus, P-grade, and Traina WFMS. Similarly, heavyweight checkpointing is employed by GWEE and GWES. Task-level checkpointing is employed by both Pegasus and P-Grade. Proactive WFMS checkpoints at the operating system level [5].

Kepler also checkpoints at the workflow layer [3], whereas, Karajan allows checkpointing the current state of the workflow at a global level. Here, timed or program-directed checkpoints can be taken, or checkpoints can be taken automatically at preconfigured time intervals, or it can be taken manually [61]. SwinDeW-C checkpoints using a minimum time redundancy based selection strategy [62].

15.5.4 PROVENANCE

Provenance is defined as the process of metadata management. It describes the origins of data, the processes involved in its production, and the transformations it has undergone. Provenance can be associated with process(es) that aid data creation [63]. Provenance captures multiple important information like dynamics of control and data flows, their progressions, execution information, file locations, input and output information, workflow structure, form, workflow evolution, and system information [6]. Provenance is essential for interpreting data, determining its quality and ownership, providing reproducible results, optimizing efficiency, troubleshooting, and also to provide fault-tolerance [64,65].

As detailed in Fig. 15.14, provenance can be of two forms: *prospective* and *retrospective* [65]. *Prospective* provenance captures the specifications that need to be followed to generate a data product or class of data products. *Retrospective* provenance captures the executed steps similar to a detailed log of task execution. It also captures information about the execution environment used to derive a specific data product.

Provenance information is used to rerun workflows, these reruns can overcome transient system errors [66]. Provenance allows users to trace state transitions and detect the cause of inconsistencies. It is used to design recovery or undo paths from workflow fault states at the task granularity level. It is used as an effective tool to provide fault-tolerance in several WFMS.

15.5.5 RESCUE WORKFLOW

The rescue workflow technique ignores failed tasks and executes the rest of the workflow until no more forward progress can be made.

A rescue workflow description called rescue DAG containing statistical information of the failed nodes is generated, which is used for later resubmission [4]. Rescue workflow technique is used by Askalon, Kepler and DAGMan [4,62].

15.5.6 USER-DEFINED EXCEPTION HANDLING

In this fault-tolerant technique, users can specify a particular action or a predefined solution for certain task failures in a workflow. Such a technique is called user-defined exception handling [4]. This could also be used to define alternate tasks for predefined type of failures [45].

This mechanism is employed by Karajan, GWES, Proactive, and Kepler among the prominent WFMS [62,5].

15.5.7 ALTERNATE TASK

The alternate task fault-tolerant scheduling technique defines an alternative implementation of a particular task. When the predefined task fails, its alternative implementation is used for execution. This technique is particularly useful when two or more different implementations are available for a task. Each implementation has different execution characteristics but take the same input and produce same outputs. For example, there could be a task with two implementations, where one is less memory or compute intensive but unreliable, while the alternate implementation is memory intensive or compute intensive but more reliable. In such cases, the later implementation can be used as an alternative task.

This technique is also useful to semantically undo the effect of a failed task, that is, alternate tasks can be used to clean up the states and data of a partially executed failed task [4,45].

15.5.8 FAILURE MASKING

Failure masking fault-tolerant technique ensures service availability, despite failures in tasks or resources [57]. This is typically achieved by redundancy, and in the event of failure the services are provided by the active (i.e., surviving) tasks or resources masking failures. Masking can be of two forms: *hierarchical group masking* and *flat group masking*. (See Fig. 15.15.)

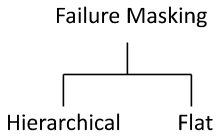

FIGURE 15.15

Forms of failure masking

Hierarchical group masking uses a coordinator to monitor the redundant components and decides which copy should replace the failed component. The major drawback of this approach is the single point of failure of the coordinator.

Flat group masking resolves this single point of failure by being symmetric. That is, the redundant components are transparent and a voting process is used to select the replacement in adversity. This approach does not have a single point of failure, but imposes more overhead to the system.

15.5.9 SLACK TIME

Task slack time represents a time window within which the task can be delayed without extending the makespan. It is intuitively related to the robustness of the schedule. Slack time is computed as the minimum spare time on any path from the considered node to the exit node of the workflow. The formal definition of slack is given by Sakellariou and Zhao in [51].

Shi et al. [10] present a robust scheduling for heterogeneous resources using slack time to schedule tasks. They present an ϵ-constraint method where robustness is an objective and deadline is a constraint. This scheduling algorithm tries to find schedules with maximum slack time without exceeding the specified deadline. Similarly, Poola et al. [67] presented a heuristic considering heterogeneous cloud resources, they divided the workflow into partial critical paths and based on the deadline and budget added slack time to these partial critical paths. Slack time added to the schedule enables the schedule time to tolerate performance variations and failures up to a certain extent, without violating the deadline.

15.5.10 TRUST-BASED SCHEDULING ALGORITHMS

Distributed environments have uncertainties and are unreliable, added to this, some service providers may slightly violate SLAs for many reasons including profitability. Therefore, WFMS typically employ trust factor to make the schedule trustworthy. Trust is composed of many attributes including reliability, dependability, honesty, truthfulness, competence, and timeliness [68]. Including trust into workflow management significantly increases fault-tolerance and decreases failure probability of a schedule [68, 69].

Conventionally, trust models are of two types: *identity-based* and *behavior-based*. *Identity-based* trust model uses trust certificates to verify the reliabilities of components. *Behavior-based* models observe and take the cumulative historical transaction behavior and also feedback of entities to evaluate the reliability [70].

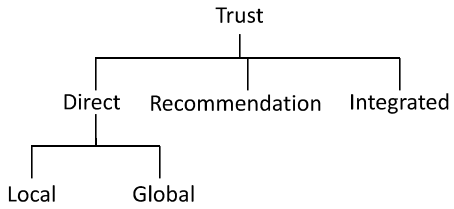

FIGURE 15.16

Methods for evaluating trust in trust-based algorithms used for fault-tolerant WFMS

Trust is evaluated by three major methods as shown in Fig. 15.16: *Direct trust*, *Recommendation Trust*, and *Integrated Trust*. *Direct trust* is derived from the historical transaction between the user and the service. Here, no third party is used to evaluate the trust of the service [70]. Direct trust can be broadly of two types *local trust* and *global trust* [71]. *Local trust* is computed based on a local system's transactions and similarly *global trust* is evaluated considering the entire global system's history. Yang et al. [69] use direct trust in their scheduling algorithm to decrease failure probability of task assignments and to improve the trustworthiness of the execution environment.

Recommendation trust is where the user consults a third party to quantify the trust of a service [70]. *Integration trust* is a combination of both direct and recommendation trust. This is usually done by a weighted approach [71]. Tan et al. [71] have proposed a reliable workflow scheduling algorithm using fuzzy technique. They propose an integrated trust metric combining direct trust and recommendation trust using a weighted approach.

Some of the drawbacks of trust models are: (i) majority of the trust models are designed for a particular environment under multiple assumptions; and (ii) trust is mostly studied in isolation without involving other system components [70].

15.6 MODELING OF FAILURES IN WORKFLOW MANAGEMENT SYSTEMS

Failure models define failure rates, frequencies and other statistically details observed in real systems, these models are used mainly in simulation and prediction systems to recreate failures. Failures can follow Poisson, exponential, Weibull, log-normal, or uniform distributions, as illustrated in Fig. 15.17. Failures can be independent or correlated. Benoit et al. [72] model resource failure through Poisson distribution, they assume failures to be statistically independent and assume a constant failure rate for each processor. Chen and Deelman [48] also assume failure to be independent but use an exponential distribution and also use a non constant failure rate. Dongarra et al's. [73] work is similar to [48], but they assume constant failure rate for each processor.

Weibull distribution is widely used in failure modeling in different ways. Litke et al. [74] use Weibull distribution to estimate the failure probability of the next assigned task for a specific resource based on the estimated execution time of each task on the resource. Plankensteiner et al. [5] use a combination of distribution to model failures. They use Weibull distribution for mean time between failure (MTBF) for clusters and to model the size of failure. Further, they use log-normal distribution to estimate the duration of failure. Rahman et al. [75] use Weibull distribution in their simulation environment

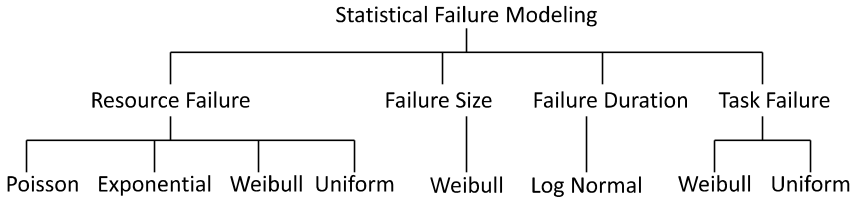

FIGURE 15.17

Distributions used for modeling failures for workflows in distributed environments

to determine whether a task execution will fail or succeed. If a task is likely to fail, they generate a random number from a uniform distribution and if that number is less than the failure probability of a resource at a particular grid, then the task is failed.

Distributions are used to evaluate reliability of tasks and resources. Wang et al. [76] uses exponential distribution to evaluate task reliability based on real-time reputation. The reputation is defined by using their task failure rate.

All the above works consider failures to be independent. However, Javadi et al. [18] consider failures to be spatial and temporally correlated. Spatial correlations of failures imply that multiple failures occur on various nodes with a specified time interval. Temporal correlation denotes skewness in failures over time. They use spherical covariance model to determine temporal failure correlation and Weibull distribution for failure modeling.

15.7 METRICS USED TO QUANTIFY FAULT-TOLERANCE

There are various metrics to measure the robustness or fault-tolerance of a workflow schedule. Each metric measures a different aspect and reports the schedule robustness based on certain constraints and assumptions. We present some prominent metrics used in the literature.

Makespan Standard Deviation reports the standard deviation of the makespan. The narrower the distribution, the better the schedule [77].

Makespan differential Entropy measures the differential entropy of the distribution: if the uncertainty is less, then the schedule is more robust [78].

Mean slack is computed by averaging task slack time, i.e., the amount of time the task can be delayed without delaying the schedule. The slack of a schedule, on the other hand, is the sum of slack times of all the tasks. Hence, the more slack a schedule has, the more failures it can tolerate. Therefore, such a schedule is more robust [78].

Probabilistic metric defines the makespan probability within two bounds. If the probability is high, then the robustness is high. This is because a higher probability indicates that the makespan is close to the average makespan [79].

Lateness likelihood metric gives the probability of the schedule to be late, where a schedule is called late if the makespan exceeds a given deadline. If the lateness likelihood is high, the robustness of the schedule is low [10].

Reliability of a compute service during a given time is defined as follows:

$$Reliability = (1 - (numFailure/n)) * mttf, \tag{15.1}$$

where *numFailure* is the number of failures experienced by the users, n is the number of users, and *mttf* is the promised mean time to failure [80].

Workflow Failure Ratio is the percentage of failed workflows due to one or more task failures [81].

Request Rejection Ratio is the ratio of number of rejected requests to the total requests [81].

Workflow Success Probability is given as a product of the success probabilities of individual tasks [28].

Standard Length Ratio indicates the performance of the workflow. It is the ratio of turnaround time to the critical path time including the communication time between tasks. Turnaround time is the workflows' running time. A lower value of this metric signifies better performance [28].

Trust metric presents the trustworthiness of a particular resource. It is given by the following equation:

$$Trust(S_i) = w_i * DT(S_i) + (1 - w_i) * RT(S_i), \tag{15.2}$$

where $DT(S_i)$ is the direct trust based on historical experiences of the ith service, $RT(S_i)$ is the recommendation trust by other users, and w_i is the weight of $DT(S_i)$ and $RT(S_i)$ for the ith service [71].

Failure probability (R_p) is the likelihood of the workflow to fail before the given deadline [10,67], which can be formulated as

$$R_p = (TotalRun - FailedRun)/(TotalRun), \tag{15.3}$$

where *TotalRun* is number of times the experiment was conducted and *FailedRun* is number of times the constraint, $finish_{t_n} \leqslant D$, was violated. Here, D is the deadline of the workflow and $finish_{t_n}$ is the workflow elapsed time.

Tolerance time (R_t) is the amount of time a workflow can be delayed without violating the deadline constraint. This provides an intuitive measurement of robustness given the same schedule and resource to task mapping, expressing the amount of uncertainties it can further withstand. It is given by

$$R_t = D - finish_{t_n}. \tag{15.4}$$

15.8 SURVEY OF WORKFLOW MANAGEMENT SYSTEMS AND FRAMEWORKS

This section provides a detailed view of the state-of-the-art WFMSs and also provide information about the different fault-tolerant techniques used, as described in Section 15.5. These WFMSs are summarized in Table 15.1.

15.8.1 ASKALON

Askalon [52] is a WFMS developed at the University of Innsbruck, Austria. It facilitates the development and optimization of applications on grid computing environments [52,4]. The system architecture

Table 15.1 Features, provenance information and fault-tolerant strategies of workflow management systems

WFMS	Features	Provenance	Fault-tolerant strategy
Askalon University of Innsbruck, Austria. http://www.dps.uibk.ac.at/projects/askalon/	• Service Oriented Architecture • Single Access User Portal • UML Workflow Editor • X.509 certificates support • Amazon EC2 API support • Grids and clouds	N/A	Resubmission, replication, checkpointing/restart, migration, user-defined exception, rescue workflow.
Pegasus USC Information Sciences Institute and the University of Wisconsin Madison. http://pegasus.isi.edu/	• Portability/Reuse • Performance and reliability • Scalability • Provenance • Data Management • Desktops, clusters, grids, and clouds	Keeps track of data locations, data results, and software used with its parameters.	Task Resubmission, Workflow Resubmission, workflow-level checkpointing, alternative data sources, rescue workflow.
Triana Cardiff University, United Kingdom.	• Modular java workflow environment • Job queuing • Comprehensive toolbox libraries • Grids and clouds	N/A	Light-weight checkpointing and restart of services are supported at the workflow level. Resubmissions are supported at the task level by the workflow engine, and alternate task technique is also employed.
Unicore 6 Collaboration between German research institutions and industries.	• Support for virtual organizations, X.509 certificates • Improved data management through DataFinder • Supports for each loops and iteration over file-sets • Grids and cluster	N/A	Resubmission and reliability measurement of task and workflows are supported.
Keplar UC Davis, UC Santa Barbara, and UC San Diego. https://kepler-project.org/	• Independently Extensible, Reliable, open and a comprehensive system • Supports multidisciplinary applications • Grids, clusters, and clouds	Data and process provenance information is recorded.	Resubmissions, checkpointing, alternative versions, error-state and user-defined exception handling mechanisms to address issues are employed.
Cloudbus WF Engine The University of Melbourne, Australia. http://cloudbus.org/workflow/	• Easy to use Graphical editor • User-friendly portal for discovery, monitoring and scheduling • Grids, clusters, and clouds	Provenance information of data is recorded.	Failure are handled by resubmitting the tasks to resources.

Table 15.1 (*Continued*)

WFMS	Features	Provenance	Fault-tolerant Strategy
Taverna Created by the myGrid team.	• Capable of performing iterations and looping • Supports data streaming • Grids, clusters, and clouds	Provenance suite records service invocations and workflow results both intermediate and final.	Resubmission and alternate resources.
e-Science Central Newcastle University, United Kingdom. http://www.esciencecentral.co.uk/	• Easy and efficient access through web browser • Provides APIs for external applications • All data are versioned • Private and public clouds	e-SC provenance service collects information regarding all system events.	Provides fine grained security control modeled around groups and user-to-user connections.
SwinDeW-C Swinburne University of Technology, Australia.	• Cloud based peer-to-peer WFMS • Web portal allows users to access entire WFMS • Clouds	data provenance is recorded during workflow execution	Checkpointing is employed. QoS management components includes performance management, data management and security management.

of it consists of the following components: (i) *Scheduler*, which maps single or multiple workflows tasks onto the grid; (ii) *Enactment Engine*, which ensures reliable and fault-tolerant execution of applications; (iii) *Resource Manager*, which is responsible for negotiation, reservation, allocation of resources and automatic deployment of services. It also shields the user from low-level grid middleware technology; (iv) *Performance Analysis*, which supports automatic instrumentation and bottleneck detection (e.g., excessive synchronization, communication, load imbalance, inefficiency, or nonscalability) within the grid; and (v) *Performance Prediction service*, which estimates execution times of workflow activities through a training phase and statistical methods based on a combination of historical data obtained from the training phase and analytical models [82,52].

Askalon uses an XML-based workflow language called AGWL for workflow orchestration. It can be used to specify DAG-constructs, parallel loops and conditional statements such as switch and if/then/else. AGWL can express sequence, parallelism choice and iteration workflow structures. Askalon uses a graphical interface called Teuta to support the graphical specification of grid workflow applications based on the UML activity diagram [82,52].

Askalon can detect faults at the following levels: (i) Hardware level, which includes machine crashes and network failures; (ii) OS level, which comprises exceeded disk quota, out of disk space, and file not found errors; (iii) Middleware-level, accounting for failed authentication, failed job-submission, unreachable services and file staging failures; and (iv) Workflow level, collecting unavailable input data, data movement faults. However, the system cannot detect task-level faults. Further, the system can recover from the following faults at different levels: (i) Hardware level, which includes machine crashes and network failures; (ii) OS level, dealing with exceeded disk quota, out of disk space; (iii) Middleware-level, which includes failed job-submission; and (iv) Workflow level, containing data movement faults. Nonetheless, it does not recover from task level faults and user-defined exceptions. Fault-tolerant techniques like checkpointing, migration, restart, retry, and replication are employed to recover from these faults [5,82,52].

15.8.2 PEGASUS

It is a project of the USC Information Sciences Institute and the Computer Science department at the University of Wisconsin Madison, United States. Pegasus enables scientists to construct workflows in abstract terms by automatically mapping the high-level workflow descriptions onto distributed infrastructures (e.g., Condor, Globus, or Amazon EC2). Multiple workflow applications can be executed in this WFMS [83].

Workflows can be described using DAX a DAG XML description. The abstract workflow describes application components and their dependencies in the form of a DAG [47].

Workflow application can be executed in variety of target platforms including local machine, clusters, grids and clouds. The WFMS executes jobs, manages data, monitors execution and handles failures. Pegasus WFMS has five major components: (i) *Mapper*, which generates an executable workflow from an abstract workflow. It also restructures the workflow to maximize performance. It further adds transformations aiding in data management and provenance generation; (ii) *Local Execution Engine*, which submits jobs to the local scheduling queue by managing dependencies and changing the state; (iii) *Job Scheduler*, which schedules and manages individual jobs on local and remote resources; (iv) *Remote Execution Engine*, which manages execution of one or more tasks on one or more remote nodes; and (v) *Monitoring Component*, which monitors the workflow execution. It records the

tasks logs, performance and provenance information in a workflow database. It notifies events such as failures, success, and statuses [84].

Pegasus stores and queries information about the environment, such as storage systems, compute nodes, data location, through various catalogs. Pegasus discovers logical files using the Replica Catalog. It looks up various user executables and binaries in Transformation Catalog. Site Catalog is used to locate computational and storage resources [84,47].

Pegasus has its own lightweight job monitoring service called Kickstart. The mapper embeds all jobs with Kickstart [84]. This helps in getting runtime provenance and performance information of the job. This information is further used for monitoring the application.

Resource selection is done using the knowledge of available resources, their characteristics and the location of the input data. Pegasus supports pluggable components where a customized approach for site selection can be performed. It has few choices of selection algorithms, such as random, round-robin, and min-min.

Pegasus can handle failures dynamically at various levels building on the features of DAGMan and HTCondor. It is equipped to detect and recover from faults. It can detect faults at the following levels: At the Hardware and Operating System levels, it can detect exceeding CPU time limit and file nonexistence. At the level of Middleware, it detects authentication, file staging, and job submission faults. At Task and Workflow levels job crashes and input unavailability are detected. DAGMan helps recover the following failures at different levels: at Hardware level, it can recover from machine crashes and network failures by automatically resubmitting. Middleware faults detected can also be recovered. Data movement faults can also be treated with recovery at task and workflow level. At Workflow level, redundancy is used and lightweight checkpoints are supported [5,84]. If a job fails more than the set number of retries, then the job is marked as a fatal failure. When a workflow fails due to such failures, the DAGMan writes a rescue workflow. The rescue workflow is similar to the original DAG without the fatal failure nodes. This workflow will start from the point of failure. Users can also replan the workflow, in case of workflow failures and move the computation left to an alternate resource. Pegasus uses retries, resubmissions, and checkpointing to achieve fault-tolerance [84].

Monitoring and debugging is also done to equip users to track and monitor their workflows. Three different logs are generated which are used to collect and process data [84]: (i) Pegasus Mapper Log helps relate the information about the abstract workflow from the executable workflow allowing users to correlate user-provided tasks to the jobs created by Pegasus. (ii) Local workflow execution engine logs contain status of each job of the workflow. (iii) Job logs capture provenance information about each job. It contains fine-grained execution statistics for each task. It also includes a web dashboard to facilitate monitoring [84].

15.8.3 TRIANA

Triana [85] is a data-flow system developed at Cardiff University, United Kingdom. It is a combination of an intuitive graphical interface with data analysis tools. It aims to support applications on multiple environments, such as peer-to-peer and grid computing. Triana allows users to integrate their own middleware and services besides providing a vast library of prewritten tools. These tools can be used in a drag-and-drop fashion to orchestrate a workflow.

Triana addresses fault-tolerance in a user-driven and interactive manner. When faults occur, the workflow is halted, displaying a warning, and allowing the user to rectify. At the hardware level,

machine crashes and network errors are detected. Missing files and other faults are detected by the workflow engine at the operating system level. Except deadlock and memory leaks that cannot be detected at the middleware and the task level, all other faults can be detected. In the workflow level, data movement and input availability errors are detected. Lightweight checkpointing and restart of services are supported at the workflow level. Retires, alternate task creations, and restarts are supported at the task level by the workflow engine [5].

15.8.4 UNICORE 6

Unicore [86] is a European grid technology developed by collaboration between German research institutions and industries. Its main objective is to access distributed resources in a seamless, secure, and intuitive way. The architecture of UNICORE is divided into three layers, namely, *client layer*, *service layer*, and *systems layer*. In the client layer, various clients, like UNICORE Rich Client (graphical interface), UNICORE command-line (UCC) interface, and High Level API (HiLA), a programming API, are available.

The service layer contains all the vital services and components. This layers has services to maintain a single site or multiple sites. Finally, the system layer has the Target System Interface (TSI) between the UNICORE and the low-level resources. Recently added functionalities to UNICORE 6 contains support for virtual organizations, interactive access based on X.509 certificates using Shibboleth, and improved data management through the integration of DataFinder. GridBeans and JavaGAT help users to support their applications further. UNICORE 6 also introduces for-each-loops and iteration over file-sets in addition to existing workflow constructs. It also supports resubmission and reliability measurement for task and workflows. Added to these new monitoring tools, availability and service functionality are also improved.

15.8.5 KEPLER

The Kepler system [87,88,3] is developed and maintained by the cross-project collaboration consisting of several key institutions: UC Davis, UC Santa Barbara, and UC San Diego. Kepler system allows scientists to exchange, archive, version, and execute their workflows.

Kepler is built on Ptolemy, a dataflow-oriented system. It focuses on an actor-oriented modeling with multiple component interaction semantics. Kepler can perform both static and dynamic checking on workflow and data. Scientists can prototype workflows before the actual implementation. Kepler system provides web service extensions to instantiate any workflow operation. Their grid service enables scientists to use grid resources over the Internet for a distributed workflow. It further supports foreign language interfaces via the Java Native Interface (JNI), giving users the benefits to use existing code and tools. Through Kepler users can link semantically compatible but syntactically incompatible services together (using XSLT, Xquery, etc.). Kepler supports heterogeneous data and file formats through Ecological Metadata Language (EML) ingestion. Fault-tolerance is employed through retries, checkpointing, and alternative versions.

15.8.6 CLOUDBUS WORKFLOW MANAGEMENT SYSTEM

The WFMS [89–91] developed at The University of Melbourne provides an efficient management technique for distributed resources. It aids users by enabling their applications to be represented as a

workflow and then execute them on the cloud platform from a higher level of abstraction. The WMS is equipped with an easy-to-use graphical workflow editor for application composition and modification, an XML-based workflow language for structured representation. It further includes a user-friendly portal with discovery, monitoring, and scheduling components.

Workflow monitor of the WFMS enables users to view the status of each task, they can also view the resource and the site where the task is executed. It also provides the failure history of each task. The workflow engine contains workflow language parser, resource discovery, dispatcher, data management, and scheduler. Tuple space model, event-driven approach, and subscription approach make WMS flexible and loosely coupled in design. Failures are handled by resubmitting the tasks to resources without a failure history for such tasks. WMS uses either Aneka [92] and/or Broker [93] to manage applications running on distributed resources.

15.8.7 TRAVERNA

Taverna [94,95] is an open source and domain-independent WFMS created by the myGrid team. It is a suite of tools used to design and execute scientific workflows and aid in silico experimentation. Taverna engine is capable of performing iterations, looping, and data streaming. It can interact with various types of services including web services, data warehouses, grid services, cloud services, and various scripts like R, distributed command-line, or local scripts.

The Traverna server allows workflows to be executed in distributed infrastructures like clusters, grids and clouds. The server has an interface called *Taverna Player* through which users can execute workflows from web browsers or through third-party clients. *Taverna Provenance suite* records service invocations and workflow results both intermediate and final. It also supports pluggable architecture that facilitates extensions and contributions to the core functionalities. Here, retries and alternate resources are used to mitigate failures.

15.8.8 THE E-SCIENCE CENTRAL (E-SC)

The e-Science Central [96] was created in 2008 as a cloud data processing system for e-Science projects. It can be deployed on both private and public clouds. Scientists can upload data, edit, run workflows, and share results using a Web Browser. It also provides an application programming interface through which external application can use the platform's functionality.

The e-SC facilitates data storage management, tools for data analysis, automation tools, and also controlled data sharing. All data are versioned and support reproduction of experiments, aiding investigation into data changes, and their analysis.

The e-SC provenance service collects information regarding all system events and this provenance data model is based on the *Open Provenance Model* (OPM) standard. It also provides fine grained security control modeled around groups and user-to-user connections.

15.8.9 SWINDEW-C

Swinburne Decentralized Workflow for cloud (SwinDeW-C) [62] is a cloud based peer-to-peer WFMS developed at Swinburne University of Technology, Australia. It is developed based on their earlier project for grid called SwinDeW-G. It is built on SwinCloud infrastructure that offers unified com-

puting and storage resources. The architecture of SwinDeW-C can be mapped into four basic layers: *application layer*, *platform layer*, *unified resource layer*, and *fabric layer*.

In SwinDeW-C users should provide workflow specification consisting of task definitions, process structures, and QoS constraints. SwinDeW-C supports two types of peers: An ordinary SwinDeW-C peer is a cloud service node with software service deployed on a virtual machine; and SwinDeW-C coordinator peers, are special nodes with QoS, data, and security management components. The cloud workflow specification is submitted to any coordinated peer, which will evaluate the QoS requirement and determine its acceptance through a negotiation process. A coordinated peer is setup within every service provider. It also has pricing and auditing components. All peers that reside in a service provider communicate with its coordinated peer for resource provisioning. Here, each task is executed by a SwinDeW-C peer during the run-time stage.

SwinDeW-C also allows virtual machines to be created with public clouds providers, such as Amazon, Google, and Microsoft. Checkpointing is employed for providing reliability. Additionally, QoS management components including performance management, data management, and security management are integrated into the coordinated peers.

15.8.10 BIG DATA WORKFLOW FRAMEWORKS: MAPREDUCE, HADOOP, AND SPARK

Recently, big data analytics has gained considerable attention both in academia and industry. Big data analytics is heavily reliant on tools developed for such analytics. In fact, these tools implement a specific form of workflows, known as MapReduce [97].

MapReduce framework is a runtime system for processing big data workflows. The framework usually runs on a dedicated platform (e.g., a cluster). There are currently two major implementations of the MapReduce framework. The original implementation with a proprietary license was developed by Google [97]. After that, Hadoop framework [98] was developed as an open-source product by Yahoo! and widely applied for big data processing.

The MapReduce framework is based on two main input functions, *Map* and *Reduce* that are implemented by the programmer. Each of these functions is executed in parallel on large-scale data across the available computational resources. Map and Reduce collectively form a usually huge workflow to process large datasets. The MapReduce storage functionality for storing input, intermediate, and output data is supported by distributed file systems developed specifically for this framework, such as Hadoop Distributed File System (HDFS) [99] and Google File System (GFS) [100].

More specifically, every MapReduce program is composed of three subsequent phases namely, *Map*, *Shuffle*, and *Reduce*. In the Map phase, the Map function implemented by the user is executed on the input data across the computational resources. The input data is partitioned into chunks and stored in a distributed file system (e.g., HDFS). Each Map task loads some chunks of data from the distributed file system and produces intermediate data that are stored locally on the worker machines. Then, the intermediate data are fed into the Reduce phase. That is, the intermediate data are partitioned to some chunks and processed by the Reduce function, in parallel.

Distributing the intermediate data across computational resources for parallel Reduce processing is called Shuffling. The distribution of intermediate data is accomplished in an all-to-all manner that imposes a communication overhead and often is the bottleneck. Once the intermediate data are distributed, the user-defined Reduce function is executed and the output of the MapReduce is produced. It is also possible to have a chain of MapReduce workflows (a.k.a. multistage MapReduce), such as

Yahoo! WebMap [101]. In these workflows, the output of a MapReduce workflow is the intermediate data for the next MapReduce workflow.

Spark [42] is a framework developed at UC Berkeley and is being utilized for research and production applications. Spark offers a general-purpose programming interface in the Scala programming language [102] for interactive and in-memory data mining across clusters with large datasets. Spark has proven to be faster than Hadoop for iterative applications.

MapReduce has been designed to tolerate faults that commonly occur at large scale infrastructures where there are thousands of computers and hundreds of other devices such as network switches, routers, and power units. Google and Hadoop MapReduce can tolerate crashes of Map and Reduce tasks. If one of these tasks stops, it is detected and a new instance of the same task is launched. In addition, data are stored along with their checksum on disks that enables corruption detection. MapReduce [97] uses a log-based approach for fault tolerance. That is, output of the Map and Reduce phases are logged to the disk [103] (e.g., a local disk or a distributed file system). In this case, if a Map task fails then it is reexecuted with the same partition of data. In case of failure in the Reduce phase, the key/value pairs for that failed Reducer have to be regenerated.

15.8.11 OTHER WORKFLOW MANAGEMENT SYSTEMS

WFMSs are in abundance that can schedule workflows on distributed environments. These WFMS primarily schedule application on clusters and grids. Karajan [61] is one such WFMS, which was implemented to overcome the shortcoming of GridAnt [104]. It was developed at the Argonne National Laboratory. Karajan is based on the definition of hierarchical workflow components.

Imperial College e-Science Network Infrastructure (ICENI) [105] was developed at London e-science center, which provides a component-based grid-middleware. GridFlow [106], Grid Workflow Execution Engine [107], P-Grade [108], Chemomentum [109] are other WFMS that schedule workflow applications on grid platforms. Each of these workflow engine have their own unique properties and have different architectures supported by a wide variety of tools and software.

15.9 TOOLS AND SUPPORT SYSTEMS

15.9.1 DATA MANAGEMENT TOOLS

Workflow enactment engine need to move data from compute nodes to storage resources and also from one node to another. Kepler uses GridFTP [87] to move files, to fetch files from remote locations. Unicore uses a data management system called DataFinder [110]. It provides with management of data objects and hides the specifics of storage systems by abstracting the data management concepts. For archival of data Tivoli Storage Manager[1] could be used. It reduces backup and recovery infrastructure. It can also back up into the cloud with openstack and vCloud integrations. Traditional protocols like HTTP, HTTPS, SFTP are also used for data movement.

[1] http://www-03.ibm.com/software/products/en/tivostormana/.

15.9.2 SECURITY AND FAULT-TOLERANCE MANAGEMENT TOOLS

In SwinDeW-C secure communications are ensured through GnuPG,[2] which is a free implementation of OpenPGP. Globus uses the X.509 certificates, an established secure data format. These certificates can be shared among public key based software. Unicore 6 employs an interactive access based on X.509 certificates called Shibboleth[3] that enables Single Sign-On as well as authentication and authorization. The International Grid Trust Federation[4] (IGTF) is a trust service provider that establishes common policies and guidelines. Similarly, The European Grid Trust project[5] provides new security services for applications using GRID middleware layer.

Access control to services can be attained through access control lists (ACLs), which can be attached to data items so that privileges for specific users and groups can be managed. DAGMan offers fault-tolerance to Pegasus through its rescue DAG. Additionally, provenance plays an important role in fault-tolerance. Most WFMS use Open Provenance Model format[6] and the W3C PROV model[7] to achieve and manage provenance information.

15.9.3 CLOUD DEVELOPMENT TOOLS

Infrastructure resources are offered by public and private clouds. Public clouds are offered by many providers like Amazon AWS, Google Compute Engine, Microsoft Azure, IBM cloud, and many others. Private clouds could be built using Openstack, Eucalyptus, and VMware, to name a few. Cloud providers offer many storage solutions that can be used by WFMSs. Some of the storage solutions offered are Amazon S3, Google's BigTable, and the Microsoft Azure Storage. Oracle also offers a cloud based database as a service for business.

Amazon through its Amazon Simple Workflow (SWF)[8] provides a fully-managed task coordinator through which developers can build, run, and scale jobs. Chaos Monkey[9] is a free service that randomly terminates resources in your cloud infrastructures. This helps test the system for failures and help develop fault-tolerant systems in cloud.

15.9.4 SUPPORT SYSTEMS

myExperiment [111] is a social network environment for e-Scientist developed by a joint team from the universities of Southampton, Manchester, and Oxford. It provides a platform to discuss issues in development, to share workflows and reuse other workflows. It is a workflow warehouse and a gateway to established environments.

---

[2] https://www.gnupg.org/.
[3] http://www.internet2.edu/products-services/trust-identity-middleware/shibboleth/.
[4] http://www.igtf.net/.
[5] http://www.gridtrust.eu/gridtrust/.
[6] http://openprovenance.org/.
[7] http://www.w3.org/2011/prov.
[8] http://aws.amazon.com/swf/.
[9] https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey/.

Workflow Generator [112], created by Pegasus provides synthetic workflow examples with their detailed characteristics. They also provide a synthetic workflow generator and traces and execution logs from real workflows.

Failure Trace Archive [113] is a public repository of availability traces of parallel and distributed systems. It also provides tools for their analysis. This will be useful in developing fault-tolerant workflow schedulers.

15.10 SUMMARY

Workflows have emerged as a paradigm for managing complex large scale data analytics and computation. They are largely used in distributed environments such as, grids and clouds to execute their computational tasks. Fault-tolerance is crucial for such large scale complex applications running on failure-prone distributed environments. Given the large body of research in this area, in this chapter, we provided a comprehensive view on fault-tolerance for workflows in various distributed environments.

In particular, this chapter provides a detailed understanding of faults from a generic viewpoint (e.g., transient, intermittent, and permanent) and a processor viewpoint (such as crash, fail-stop, and byzantine). It also describes techniques such as replication, resubmission, checkpointing, provenance, rescue-workflow, exception handling, alternate task, failure masking, slack time, and trust-based approaches used to resolve these faults by which, a transparent and seamless experience to workflow users can be offered.

Apart from the fault-tolerant techniques, this chapter provides an insight into numerous failure models and metrics. Metrics range from makespan oriented, probabilistic, reliability based, and trust-based among others. These metrics inform us about the quality of the schedule and quantify fault-tolerance of a schedule.

Prominent WFMSs are detailed and positioned with respect to their features, characteristics, and uniqueness. Lastly, tools such as those for describing workflow languages, data-management, security, and fault-tolerance, and tools that aid in cloud development and support systems (including social networking environments and workflow generators) are introduced.

In effect, the stance of this chapter is helpful for developers and researchers working in the area of workflow management systems, as it identifies strength and weaknesses in this field and proposes future directions. This chapter provides a holistic view of fault-tolerance in WFMSs and techniques employed by different existing systems. The chapter also identifies the research trends and provides recommendations on future research areas in the area of fault-tolerant workflow management systems.

REFERENCES

[1] G. Juve, E. Deelman, Scientific workflows and clouds, Crossroads 16 (3) (2010) 14–18.

[2] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers, Examining the challenges of scientific workflows, Computer 40 (12) (2007) 24–32.

[3] P. Mouallem, D. Crawl, I. Altintas, M. Vouk, U. Yildiz, A fault-tolerance architecture for Kepler-based distributed scientific workflows, in: M. Gertz, B. Ludäscher (Eds.), Scientific and Statistical Database Management, in: Lecture Notes in Computer Science, vol. 6187, 2010, pp. 452–460.

[4] J. Yu, R. Buyya, A taxonomy of scientific workflow systems for grid computing, SIGMOD Rec. 34 (3) (2005) 44–49.
[5] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertesz, P. Kacsuk, Fault detection, prevention and recovery in current grid workflow systems, in: Grid and Services Evolution, 2009, pp. 1–13.
[6] M.A. Vouk, Cloud computing – issues, research and implementations, CIT, J. Comput. Inf. Technol. 16 (4) (2008) 235–246.
[7] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, New York, 1979, USA.
[8] Y. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (4) (1999) 406–471.
[9] J. Yu, R. Buyya, K. Ramamohanarao, Workflow scheduling algorithms for grid computing, in: F. Xhafa, A. Abraham (Eds.), Metaheuristics for Scheduling in Distributed Computing Environments, Stud. Comput. Intell. 146 (2008) 173–214.
[10] Z. Shi, E. Jeannot, J. Dongarra, Robust task scheduling in non-deterministic heterogeneous computing systems, in: IEEE International Conference on Cluster Computing, 2006, IEEE, 2006, pp. 1–10.
[11] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Gener. Comput. Syst. 29 (2013) 682–692.
[12] V. Leon, S. Wu, H. Robert, Robustness measures and robust scheduling for job shops, IIE Trans. 26 (5) (1994) 32–43.
[13] W. Herroelen, R. Leus, Project scheduling under uncertainty: survey and research potentials, Eur. J. Oper. Res. 165 (2) (2005) 289–306.
[14] J. Smith, H. Siegel, A. Maciejewski, Robust Resource Allocation in Heterogeneous Parallel and Distributed Computing Systems, Wiley Online Library, 2008, USA.
[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys'07, 2007, pp. 59–72.
[16] S.Y. Ko, I. Hoque, B. Cho, I. Gupta, Making cloud intermediate data fault-tolerant, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10, 2010, pp. 181–192.
[17] J. Dean, Experiences with MapReduce, an abstraction for large-scale computation, in: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, PACT'06, 2006.
[18] B. Javadi, J. Abawajy, R. Buyya, Failure-aware resource provisioning for hybrid cloud infrastructure, J. Parallel Distrib. Comput. 72 (10) (2012) 1318–1331.
[19] F. Gärtner, Fundamentals of fault-tolerant distributed computing in asynchronous environments, ACM Comput. Surv. 31 (1) (1999) 1–26.
[20] M. Lackovic, D. Talia, R. Tolosana-Calasanz, J. Banares, O. Rana, A taxonomy for the analysis of scientific workflow faults, in: Proceedings of the 13th IEEE International Conference on Computational Science and Engineering, 2010, pp. 398–403.
[21] A. Benoit, L.-C. Canon, E. Jeannot, Y. Robert, Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms, J. Sched. 15 (5) (2012) 615–627.
[22] R.D. Schlichting, F.B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans. Comput. Syst. 1 (3) (1983) 222–238.
[23] C. Dabrowski, Reliability in grid computing systems, Concurr. Comput., Pract. Exp. 21 (8) (2009) 927–959.
[24] W. Cirne, F. Brasileiro, D. Paranhos, L. Goes, W. Voorsluys, On the efficacy, efficiency and emergent behavior of task replication in large distributed systems, Parallel Comput. 33 (3) (2007) 213–234.
[25] A. Benoit, M. Hakem, Y. Robert, Fault tolerant scheduling of precedence task graphs on heterogeneous platforms, in: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, 2008, pp. 1–8.
[26] D. Mosse, R. Melhem, S. Ghosh, Analysis of a fault-tolerant multiprocessor scheduling algorithm, in: Proceedings of the 24th International Symposium on Fault-Tolerant Computing, FTCS-24, 1994, pp. 16–25.
[27] G. Kandaswamy, A. Mandal, D. Reed, Fault tolerance and recovery of scientific workflows on computational grids, in: Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid, 2008, pp. 777–782.
[28] Y. Zhang, A. Mandal, C. Koelbel, K. Cooper, Combined fault tolerance and scheduling techniques for workflow applications on computational grids, in: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID, 2009, pp. 244–251.
[29] K. Hashimoto, T. Tsuchiya, T. Kikuno, Effective scheduling of duplicated tasks for fault tolerance in multiprocessor systems, IEICE Trans. Inf. Syst. 85 (3) (2002) 525–534.

[30] A. Chervenak, E. Deelman, M. Livny, M.H. Su, R. Schuler, S. Bharathi, G. Mehta, K. Vahi, Data placement for scientific applications in distributed environments, in: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, GRID'07, 2007, pp. 267–274.
[31] S. Darbha, D. Agrawal, A task duplication based optimal scheduling algorithm for variable execution time tasks, in: Proceedings of the International Conference on Parallel Processing, vol. 1, in: ICPP, vol. 2, 1994, pp. 52–56.
[32] S. Ranaweera, D. Agrawal, A task duplication based scheduling algorithm for heterogeneous systems, in: Proceedings of the 14th International Parallel and Distributed Processing Symposium, 2000, pp. 445–450.
[33] A. Dogan, F. Ozguner, LDBS: a duplication based scheduling algorithm for heterogeneous computing systems, in: Proceedings of the International Conference on Parallel Processing, 2002, pp. 352–359.
[34] X. Tang, X. Li, G. Liao, R. Li, List scheduling with duplication for heterogeneous computing systems, J. Parallel Distrib. Comput. 70 (4) (2010) 323–329.
[35] R. Calheiros, R. Buyya, Meeting deadlines of scientific workflows in public clouds with tasks replication, IEEE Trans. Parallel Distrib. Syst. PP (99) (2013) 1.
[36] I. Brandic, D. Music, S. Dustdar, Service mediation and negotiation bootstrapping as first achievements towards self-adaptable grid and cloud services, in: Proceedings of the 6th International Conference Industry Session on Grids Meets Autonomic Computing, GMAC '09, New York, NY, USA, 2009, pp. 1–8.
[37] D. Yuan, L. Cui, X. Liu, Cloud data management for scientific workflows: Research issues, methodologies, and state-of-the-art, in: Proceedings of the 10th International Conference on Semantics, Knowledge and Grids (SKG), 2014, pp. 21–28.
[38] W. Li, Y. Yang, D. Yuan, A novel cost-effective dynamic data replication strategy for reliability in cloud data centres, in: Proceedings of the 9th IEEE International Conference on Dependable, Autonomic and Secure Computing, DASC'11, 2011, pp. 496–502.
[39] J. Dean, Software engineering advice from building large-scale distributed systems, http://research.google.com/people/jeff/stanford-295-talk.pdf, 2007.
[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, 2012, pp. 2–12.
[41] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'10, 2010, pp. 135–146.
[42] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, 2010, pp. 10–15.
[43] J. Li, M. Humphrey, Y. Cheah, Y. Ryu, D. Agarwal, K. Jackson, C. van Ingen, Fault tolerance and scaling in e-science cloud applications: observations from the continuing development of MODISAzure, in: Proceedings of the IEEE 6th International Conference on e-Science (e-Science), 2010, pp. 246–253.
[44] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, A. YarKhan, New grid scheduling and rescheduling methods in the grads project, Int. J. Parallel Program. 33 (2–3) (2005) 209–229.
[45] S. Hwang, C. Kesselman, Grid workflow: a flexible failure handling framework for the grid, in: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, 2003, pp. 126–137.
[46] D. Poola, K. Ramamohanarao, R. Buyya, Fault-tolerant workflow scheduling using spot instances on clouds, in: Proceedings of the International Conference on Computational Science in the Procedia Computer Science, in: International Conference on Computational Science, vol. 29, 2014, pp. 523–533.
[47] E. Deelman, G. Singh, M.H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, D.S. Katz, Pegasus: a framework for mapping complex scientific workflows onto distributed systems, Sci. Program. 13 (3) (2005) 219–237.
[48] W. Chen, E. Deelman, Fault tolerant clustering in scientific workflows, in: Proceedings of the IEEE 8th World Congress on Services, SERVICES, 2012, pp. 9–16.
[49] Z. Yu, W. Shi, An adaptive rescheduling strategy for grid workflow applications, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2007, pp. 1–8.
[50] K. Plankensteiner, R. Prodan, Meeting soft deadlines in scientific workflows using resubmission impact, IEEE Trans. Parallel Distrib. Syst. 23 (5) (2012) 890–901.

[51] R. Sakellariou, H. Zhao, A low-cost rescheduling policy for efficient mapping of workflows on grid systems, Sci. Program. 12 (4) (2004) 253–262.
[52] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, M. Wieczorek, Askalon: a development and grid computing environment for scientific workflows, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, 2007, pp. 450–471.
[53] R. Duan, R. Prodan, T. Fahringer, Dee: a distributed fault tolerant workflow enactment engine for grid computing, in: L. Yang, O. Rana, B. Di Martino, J. Dongarra (Eds.), High Performance Computing and Communications, in: Lecture Notes in Computer Science, vol. 3726, 2005, pp. 704–716.
[54] E.N. Elnozahy, L. Alvisi, Y. Wang, D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surv. 34 (3) (2002) 375–408.
[55] J. Chen, Y. Yang, Adaptive selection of necessary and sufficient checkpoints for dynamic verification of temporal constraints in grid workflow systems, ACM Trans. Auton. Adapt. Syst. 2 (6) (2007).
[56] M.A. Salehi, A.N. Toosi, R. Buyya, Contention management in federated virtualized distributed systems: implementation and evaluation, Softw. Pract. Exp. 44 (3) (2014) 353–368.
[57] I. Egwutuoha, D. Levy, B. Selic, S. Chen, A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems, J. Supercomput. 65 (3) (2013) 1302–1326.
[58] R. Tolosana-Calasanz, J. Baòares, P. Álvarez, J. Ezpeleta, O. Rana, An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows, J. Comput. Syst. Sci. 76 (6) (2010) 403–415.
[59] M.A. Salehi, B. Javadi, R. Buyya, Resource provisioning based on preempting virtual machines in distributed systems, Concurr. Comput., Pract. Exp. 26 (2) (2014) 412–433.
[60] M.A. Salehi, J. Abawajy, R. Buyya, Taxonomy of contention management in interconnected distributed systems, in: Computer Science and Software Engineering, Computing Handbook, third edition, 2014, pp. 1–33, Chapter 57.
[61] G. von Laszewski, M. Hategan, D. Kodeboyina, Java COG kit workflow, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, 2007, pp. 340–356.
[62] X. Liu, D. Yuan, G. Zhang, J. Chen, Y. Yang, SwinDeW-C: a peer-to-peer based cloud workflow system, in: B. Furht, A. Escalante (Eds.), Handbook of Cloud Computing, 2010, pp. 309–332.
[63] Y.L. Simmhan, B. Plale, D. Gannon, A survey of data provenance in e-science, SIGMOD Rec. 34 (3) (2005) 31–36.
[64] S.B. Davidson, S.C. Boulakia, A. Eyal, B. Ludäscher, T.M. McPhillips, S. Bowers, M.K. Anand, J. Freire, Provenance in scientific workflow systems, IEEE Data Eng. Bull. 30 (4) (2007) 44–50.
[65] S.B. Davidson, J. Freire, Provenance and scientific workflows: challenges and opportunities, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'08, New York, NY, USA, 2008, pp. 1345–1350.
[66] Y. Simmhan, R. Barga, C. van Ingen, E. Lazowska, A. Szalay, Building the trident scientific workflow workbench for data management in the cloud, in: Proceedings of the 3rd International Conference on Advanced Engineering Computing and Applications in Sciences, 2009, ADVCOMP'09, 2009, pp. 41–50.
[67] D. Poola, S.K. Garg, R. Buyya, Y. Yang, K. Ramamohanarao, Robust scheduling of scientific workflows with deadline and budget constraints in clouds, in: Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA-2014), 2014, pp. 1–8.
[68] M. Wang, K. Ramamohanarao, J. Chen, Trust-based robust scheduling and runtime adaptation of scientific workflow, Concurr. Comput., Pract. Exp. 21 (16) (2009) 1982–1998.
[69] Y. Yang, X. Peng, Trust-based scheduling strategy for workflow applications in cloud environment, in: Proceedings of the 8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013, pp. 316–320.
[70] W. Li, J. Wu, Q. Zhang, K. Hu, J. Li, Trust-driven and QoS demand clustering analysis based cloud workflow scheduling strategies, Clust. Comput. 17 (3) (2014) 1013–1030.
[71] W. Tan, Y. Sun, L.X. Li, G. Lu, T. Wang, A trust service-oriented scheduling model for workflow applications in cloud computing, IEEE Syst. J. 8 (3) (2014) 868–878.
[72] A. Benoit, M. Hakem, Y. Robert, Multi-criteria scheduling of precedence task graphs on heterogeneous platforms, Comput. J. 53 (6) (2010) 772–785, http://comjnl.oxfordjournals.org/content/53/6/772.full.pdf+html.
[73] J.J. Dongarra, E. Jeannot, E. Saule, Z. Shi, Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems, in: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'07, New York, NY, USA, 2007, pp. 280–288.
[74] A. Litke, D. Skoutas, K. Tserpes, T. Varvarigou, Efficient task replication and management for adaptive fault tolerance in mobile grid environments, Future Gener. Comput. Syst. 23 (2) (2007) 163–178.

[75] M. Rahman, R. Ranjan, R. Buyya, Reputation-based dependable scheduling of workflow applications in peer-to-peer grids, Comput. Netw. 54 (18) (2010) 3341–3359.
[76] X. Wang, C.S. Yeo, R. Buyya, J. Su, Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm, Future Gener. Comput. Syst. 27 (8) (2011) 1124–1134.
[77] L. Canon, E. Jeannot, Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments, IEEE Trans. Parallel Distrib. Syst. 21 (4) (2010) 532–546.
[78] L. Bölöni, D.C. Marinescu, Robust scheduling of metaprograms, J. Sched. 5 (5) (2002) 395–412.
[79] V. Shestak, J. Smith, H. Siegel, A. Maciejewski, A stochastic approach to measuring the robustness of resource allocations in distributed systems, in: Proceedings of the International Conference on Parallel Processing, 2006, ICPP, IEEE, 2006, pp. 459–470.
[80] S.K. Garg, S. Versteeg, R. Buyya, A framework for ranking of cloud computing services, Future Gener. Comput. Syst. 29 (4) (2013) 1012–1023, Special Section: Utility and Cloud Computing.
[81] S. Adabi, A. Movaghar, A.M. Rahmani, Bi-level fuzzy based advanced reservation of cloud workflow applications on distributed grid resources, J. Supercomput. 67 (1) (2014) 175–218.
[82] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, M. Wieczorek, Askalon: a grid application development and computing environment, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, GRID'05, Washington, DC, USA, 2005, pp. 122–131.
[83] Pegasus workflow management system, https://pegasus.isi.edu/, 2014 [Online; accessed 01 December 2014].
[84] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (0) (2015) 17–35.
[85] I. Taylor, M. Shields, I. Wang, A. Harrison, The Triana workflow environment: architecture and applications, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, 2007, pp. 320–339.
[86] A. Streit, P. Bala, A. Beck-Ratzka, K. Benedyczak, S. Bergmann, R. Breu, J. Daivandy, B. Demuth, A. Eifer, A. Giesler, B. Hagemeier, S. Holl, V. Huber, N. Lamla, D. Mallmann, A. Memon, M. Memon, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, T. Schlauch, A. Schreiber, T. Soddemann, W. Ziegler, Unicore 6 – recent and future advancements, Ann. Telecommun. 65 (11–12) (2010) 757–762.
[87] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock, Kepler: an extensible system for design and execution of scientific workflows, in: Proceedings of the 16th International Conference on Scientific and Statistical Database Management, 2004, pp. 423–424.
[88] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurr. Comput., Pract. Exp. 18 (10) (2006) 1039–1065.
[89] S. Pandey, W. Voorsluys, M. Rahman, R. Buyya, J.E. Dobson, K. Chiu, A grid workflow environment for brain imaging analysis on distributed systems, Concurr. Comput., Pract. Exp. 21 (16) (2009) 2118–2139.
[90] R. Buyya, S. Pandey, C. Vecchiola, Cloudbus toolkit for market-oriented cloud computing, in: M. Jaatun, G. Zhao, C. Rong (Eds.), Cloud Computing, in: Lecture Notes in Computer Science, vol. 5931, 2009, pp. 24–44.
[91] S. Pandey, D. Karunamoorthy, R. Buyya, Workflow engine for clouds, in: Cloud Computing, 2011, pp. 321–344.
[92] C. Vecchiola, X. Chu, R. Buyya, Aneka: a software platform for .net-based cloud computing, in: High Speed and Large Scale Scientific Computing, 2009, pp. 267–295.
[93] S. Venugopal, K. Nadiminti, H. Gibbins, R. Buyya, Designing a resource broker for heterogeneous grids, Softw. Pract. Exp. 38 (8) (2008) 793–825.
[94] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054, http://bioinformatics.oxfordjournals.org/content/20/17/3045.full.pdf+html.
[95] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M.P. Balcazar Vargas, S. Sufi, C. Goble, The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud, Nucleic Acids Research, http://nar.oxfordjournals.org/content/early/2013/05/02/nar.gkt328.full.pdf+html.
[96] H. Hiden, S. Woodman, P. Watson, J. Cala, Developing cloud applications using the e-science central platform, Philos. Trans. R. Soc. Lond. A, Math. Phys. Eng. Sci. 371 (1983).
[97] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[98] C. Lam, Hadoop in Action, 1st edition, Greenwich, CT, USA, 2010.

[99] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies, MSST'10, 2010, pp. 1–10.
[100] S. Ghemawat, H. Gobioff, S. Leung, The Google file system, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP'03, 2003, pp. 29–43.
[101] O. Alaçam, M. Dalcı, A usability study of webmaps with eye tracking tool: the effects of iconic representation of information, in: Proceedings of the 13th International Conference on Human–Computer Interaction, 2009, pp. 12–21.
[102] M. Odersky, L. Spoon, B. Venners, Programming in Scala: A Comprehensive Step-by-Step Guide, 1st edition, Artima Press, Walnut Creek, California, 2008, USA.
[103] A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, C. Fetzer, A. Brito, Low-overhead fault tolerance for high-throughput data processing systems, in: Proceedings of the 31st International Conference on Distributed Computing Systems, ICDCS'11, 2011, pp. 689–699.
[104] K. Amin, G. von Laszewski, M. Hategan, N. Zaluzec, S. Hampton, A. Rossi, Gridant: a client-controllable grid workflow system, in: Proceedings of the 37th Annual Hawaii International Conference on System Sciences, 2004, p. 10.
[105] S. McGough, L. Young, A. Afzal, S. Newhouse, J. Darlington, Workflow enactment in ICENI, in: UK e-Science All Hands Meeting, 2004, pp. 894–900.
[106] J. Cao, S.A. Jarvis, S. Saini, G.R. Nudd, Gridflow: workflow management for grid computing, in: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid, 2003, pp. 198–205.
[107] E. Elmroth, F. Hernández, J. Tordsson, A light-weight grid workflow execution engine enabling client and middleware independence, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics, in: Lecture Notes in Computer Science, vol. 4967, 2008, pp. 754–761.
[108] P. Kacsuk, G. Sipos, Multi-grid, multi-user workflows in the p-grade grid portal, J. Grid Comput. 3 (3–4) (2005) 221–238.
[109] S.P. Callahan, J. Freire, E. Santos, C.E. Scheidegger, C.T. Silva, H.T. Vo, Vistrails: visualization meets data management, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'06, New York, NY, USA, 2006, pp. 745–747.
[110] T. Schlauch, A. Schreiber, DataFinder – a scientific data management solution, in: Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data, PV, Oberpfaffenhofen, Germany, 2007.
[111] C.A. Goble, D.C. De Roure, ^{my}Experiment: social networking for workflow-using e-scientists, in: Proceedings of the 2nd Workshop on Workflows in Support of Large-scale Science, WORKS'07, New York, NY, USA, 2007, pp. 1–2.
[112] Pegasus workflow generator, https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator/2014 [Online; accessed 5 December 2014].
[113] D. Kondo, B. Javadi, A. Iosup, D. Epema, The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems, in: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid, 2010, IEEE, 2010, pp. 398–407.