Fault-Avoidance Strategies for Context-Aware Schedulers in Pervasive Computing Systems

Janick Edinger, Dominik Schäfer, Christian Krupitzer, Vaskar Raychoudhury, Christian Becker
University of Mannheim
Schloss, 68161 Mannheim, Germany
Email: {janick.edinger, dominik.schaefer, christian.krupitzer, vaskar.raychoudhury, christan.becker}@uni-mannheim.de

***Abstract*—Scheduling in distributed computing systems is the process of allocating resources to a computational task. The complexity of this allocation process increases with the amount of criteria that are considered for the scheduling decision. Pervasive computing systems show a high degree of heterogeneity and dynamism. The constant joining and leaving of devices makes the system error-prone and less predictable. The involved devices differ in various properties that we subsume as their context. We argue, that these context dimensions can be used to implement fault-avoidant scheduling strategies.**

In this paper, we introduce the concept of context-aware scheduling for pervasive computing systems. The schedulers in these systems consider multiple context dimensions to avoid failing resource providers. We discuss relevant context dimensions, develop context-aware scheduling strategies and implement them into an existing distributed computing system. We show how to monitor the context dimensions and evaluate the fault-avoidant scheduling strategies in a large-scale simulation.

I. Introduction

In pervasive environments such as Aura [1], Gaia [2] or BASE [3] / PCOM [4], devices share resources. For large numbers of devices, the access has to be coordinated. Scheduling is the process of allocating resources to a computational task. Distributed schedulers do not only take local resources into account, but also offload work to remote devices that share their computational resources. In such distributed computing systems, scheduling becomes more complex, as communication between multiple machines is required and entities can join and leave the system at any time [5]. Following the taxonomy of [6], we define resource *providers* as the entities that share their computing resources with resource *consumers* that make use of these remote resources. Further, resource *controllers* perform the matchmaking between consumers and providers. Controllers typically attempt to optimize the performance in a distributed computing system which can be measured in terms of throughput or average task completion time. Scheduling strategies in stable grid and cloud environments have been extensively studied. Results are presented in [5] and [7].

In contrast to traditional grid and cloud architectures, edge computing systems also integrate end-user devices [8]. The shift from centrally managed cloud and grid resources to user-managed edge devices introduces a new level of heterogeneity into distributed computing environments. Each device has different properties, such as its hardware and software configuration, its location, and its connectivity to the network. Further, these devices do not provide any service guarantees but might stop the execution of tasks at any point in time. Fault-tolerant distributed schedulers thus have to implement dynamic rescheduling to provide a reliable execution of tasks. Previous approaches of fault-tolerant scheduling focus on fault-detection strategies and recovery mechanisms [9] or replication [10]. However, as fault-recovery introduces a delayed execution, proactive approaches that predict failures even before they occur can increase the performance of distributed computing systems [11], [12], [13], [14].

Most existing approaches take one particular property of a resource or a task into account when making scheduling decisions. However, we argue that there are plenty context dimensions in pervasive systems that can be considered to improve the quality of resource allocation. Dey et al. define context as '*[...] any information that can be used to characterize the situation of an entity.*' [15]. Besides the context of providers and consumers, the system itself has context dimensions such as the number of connected devices and the dynamism of the system. By combining the multiple context variables, dynamic and unreliable resources can be integrated into a distributed computing system.

In this paper, we introduce a context-aware, fault-avoidant scheduler for distributed computing systems. The scheduler monitors multiple context variables to predict the behavior of resource providers and implements strategies to select those providers that show a high probability to successfully execute the task. We integrate the scheduler into our Tasklet system. The Tasklet system is a middleware that offers a lightweight abstraction of computation in heterogeneous distributed computing environments. As the system does not only support stable grid and cloud resources but also includes mobile devices that might enter and leave the system spontaneously, the scheduler has to make fault-avoidant scheduling decisions.

This paper has four contributions. First, we present context dimensions of providers, consumers, and the system itself and discuss their impact on scheduling decisions. Second, we introduce the Tasklet system and define our failure model. Third, we develop scheduling strategies based on multiple context dimensions and discuss the integration of context-awareness into our system model. Fourth, we evaluate the scheduling strategies in a large-scale simulation to show how they perform in a fluctuating environment.

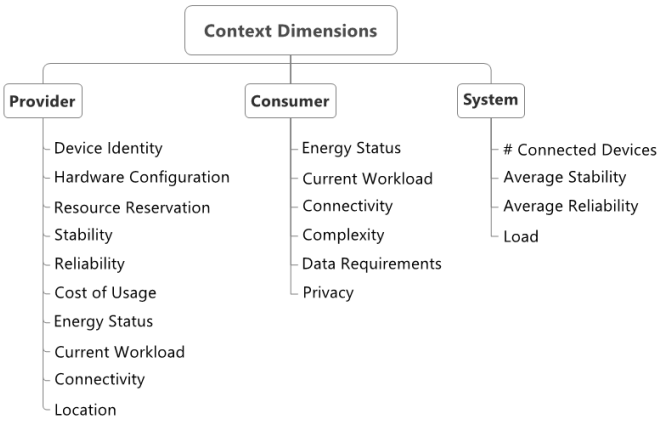

Fig. 1: Context dimensions of providers, consumers, and the system.

II. Context Model for Distributed Scheduling

Each entity in the system has its own context that might change over time. In this section, we present context dimensions for providers, consumers, and the overall system (cf. Figure 1). We discuss their impact on scheduling decisions.

A. (Mobile) Resource Providers

A unique *identifier*, such as the MAC address of a device, is the most static context variable. Consumers can select one particular device for execution of their tasks.

The *hardware configuration* of a device is rather static. The scheduler can take the hardware configuration into account for optimizing the distribution, e.g., specialized hardware such as GPUs and multi-core processors that can be useful to execute highly parallel tasks. Tasks that can benefit from parallelism, can be offloaded to providers that are well suited for parallel executions.

Resource reservation can be used for tasks with soft or hard real-time requirements. In order to guarantee that resources are available at the time they are required, devices might allow reservation. The scheduler has to maintain control over the reservations and distribute the tasks accordingly.

As devices dynamically join and leave the system the *stability* of devices is important for estimating whether a provider would successfully complete the execution of a task, such as the time the provider is already connected to the system compared to previous connection times. Long-lasting tasks can then be scheduled to the most stable providers in the system.

Reliability indicates whether a provider is reliable in finishing its tasks. It is influenced by the amount of aborted tasks. The scheduler should prefer reliable providers as this increases the probability for the completion of tasks.

Usage of resources is often associated with *costs*. Typically, costs have fixed rates, vary depending on demand and supply [16], or resources might be offered for free to either everybody or to selected consumers. As a general assumption, a good scheduling decision keeps the cost of resource usage low.

Mobile devices have a *limited battery life*. When the battery runs out, executed tasks are silently dropped. Thus the scheduler needs to consider the battery status of mobile devices.

Workload can be either caused by the owners of devices themselves or by tasks from remote applications. From the scheduling perspective a high load might result in additional delay when tasks are queued. Devices might also reject resource requests when their workload is too high.

Connectivity of resource providers varies in the dimensions of bandwidth, delay, and jitter. Whereas the connectivity of desktop computers and cloud resources remains rather stable, the connectivity of mobile devices fluctuates. For tasks that require a large amount of data, the bandwidth of resource providers should be considered for scheduling. Further, for highly responsive applications, delay and jitter are important.

In distributed systems, *location* is considered as an important context dimension [17]. When consumers want to use resources located in a particular area, e.g., in their proximity, the scheduler has to take the providers' location into account.

B. Resource Consumers

The consumer first decides between local task execution or offloading the task by taking into account its context, namely, information on its device and the task's characteristics. In the following, we discuss these context dimensions.

For mobile devices the *energy* is one of the most important bottlenecks. Offloading tasks to remote resources can save energy compared to local execution. However, the offloading process requires energy as well. Estimating whether a local execution or the offloading process can save energy is a complex task that is discussed in [18].

Another important factor is the *current workload* of the consumer. A higher workload increases the probability for offloading a task as it offers parallel execution.

Similar to the provider, *connectivity* of a consumer varies in bandwidth, delay, and jitter. A low bandwidth makes offloading less likely as the offloading process' time increases.

Complexity of a task can be measured in different units, e.g., time needed for computation of the task. A larger complexity for a task increases the probability for offloading as this may be faster than local execution.

Remote execution of tasks have different *requirements for data handling*. In cases of a high amount of data needed for a task's execution, the transfer rate between the consumer and potential providers may influence the offloading decision.

Confidential tasks or tasks that contain sensitive data require a scheduler that takes *privacy* and security into account. Privacy can be maintained by selecting only known and trusted providers. To ensure a secure transmission of tasks and results, the scheduler selects providers that offer a suitable level of security (e.g., encryption).

C. System

The *number of connected devices* is divided into the numbers of (i) providers and (ii) consumers. On the one hand, if the number of providers is high, this offers more possibilities

for scheduling as the heterogeneity is increased. On the other hand, a high number of consumer requests may lead to queuing if sufficient providers are not available.

The *average stability* of the connected devices may influence the scheduling decision. A higher stability leads to a lowered probability to choose providers that will leave the system during task execution.

A high *average reliability* indicates that a provider successfully finished Tasklets in the past. One of the objectives of the system must be to keep the average reliability as high as possible as it is an indicator for a good system performance.

The *load* of the system is measured by the amount of providers that are currently executing tasks. A higher load leads to delays as more messages are transferred within the system. Furthermore, high workload on providers result in queuing requests. Therefore, the system should try to equally distribute tasks within the system and optimize the load of providers, e.g., by assigning non-complex tasks to resource-poor providers.

III. Fault-Tolerant Distributed Computation with Tasklets

Depending on the goal of the scheduling algorithm, one or multiple context dimensions of the entities and the task are considered for the scheduling decision. In this paper, we propose a fault-avoidant scheduler that selects the most suitable resource provider for the execution of a task. We develop scheduling strategies and implement them into the *Tasklet system* – our approach for a distributed computing system.

This section first presents the Tasklet system that serves as the underlying distributed computation environment. Next, we discuss the failure model to illustrate when the execution of a task can get canceled. Subsequently, the succeeding sections present the system model for our fault-avoidant scheduler as well as corresponding strategies.

A. The Tasklet System

The goal of the Tasklet middleware is to provide an easy to use abstraction for developers to distribute the workload of computation intensive applications to local and remote resources. Tasklets are self-contained, parametrized units of computation. They can be scheduled independently from other Tasklets and any resources of the local device. Tasklets and Tasklet results are exchanged via the Tasklet middleware which runs on a variety of platforms, including desktop computers, mobile devices, and graphical processing units. To overcome the hardware heterogeneity and abstract the computation from the underlying platform, we use virtualization techniques. As a result, every Tasklet can be executed by any resource provider in the Tasklet system. Since all devices share the Tasklet middleware as common execution environment, the scheduler does not have to consider heterogeneity in platforms.

The Tasklet system consists of three entities: providers, consumers, and brokers. Resource *providers* offer their computing resources in form of virtual machines (TVMs). Resource *consumers* distribute the workload of computation intensive applications to providers. Resource *brokers* act as resource controllers and perform the matchmaking between consumers and providers. A detailed description of the Tasklet system can be found in [19]. Figure 2 shows the system model of the Tasklet system.

The scheduling of Tasklets works as follows:

1) Providers register their resources at a broker.
2) A consumer, that wants to execute a Tasklet remotely, sends an execution request to the broker.
3) The broker selects a provider for execution and returns this information to the consumer.
4) The consumer directly sends the Tasklet to the provider which executes the Tasklet on its local TVM.
5) The provider returns the result to the consumer.

The Tasklet system is a hybrid peer-to-peer system as Tasklets and Tasklet results are exchanged directly between providers and consumers whereas brokers perform the matchmaking between them. This reduces the amount of communication for the broker and avoids performance bottlenecks to make the system more scalable. The brokers in the system are connected in a peer-to-peer overlay network. They exchange information about providers and perform load balancing.

In general, the Tasklet middleware provides a best-effort execution layer: The scheduling and execution of Tasklets is implemented without providing any further guarantees. As a result, Tasklets can be dropped at any point in time and the matchmaking between Tasklets and providers is performed randomly. While this service is sufficient for some applications, other applications might require further guarantees.

Therefore, in [19], we introduce the concept of *Quality of Computation (QoC)*. This adds another layer on top of the best-effort Tasklet system. Application developers can specify QoC goals for Tasklets, such as a reliable execution. The Tasklet middleware then enforces these goals. It monitors the execution and re-initiates it in case the provider crashes during execution. Developers can use QoC goals to affect the context-aware scheduling decision. As an example, developers can request a *fast* execution. The scheduler then selects powerful providers to reduce the execution time and starts redundant executions in case one provider fails.

B. Failure Model

The execution of Tasklets can be terminated for multiple reasons. As the system includes user-managed, pervasive edge resources that might be turned off or leave the system at any time, the execution of Tasklets might fail. In the following, we show the different faults that can occur. These faults have to be taken into account for the design of a fault-avoidant scheduling mechanism.

Explicit Leave: Device owners of resource providers might shut down the application at any time. This causes the Tasklet execution to be terminated. In that case, the provider informs the consumer about the abortion of the Tasklet.

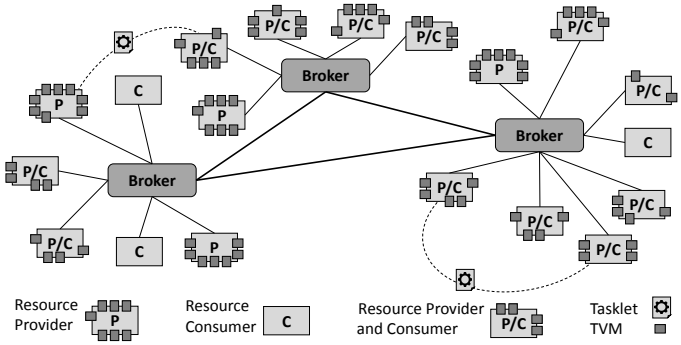

Fig. 2: The Tasklet system consists of three entities: providers (P), consumers (C), and brokers (B). To execute a Tasklet, consumers send a resource request to a broker. It selects and returns a suitable provider for execution. Tasklets and Tasklet results are exchanged directly. [19]

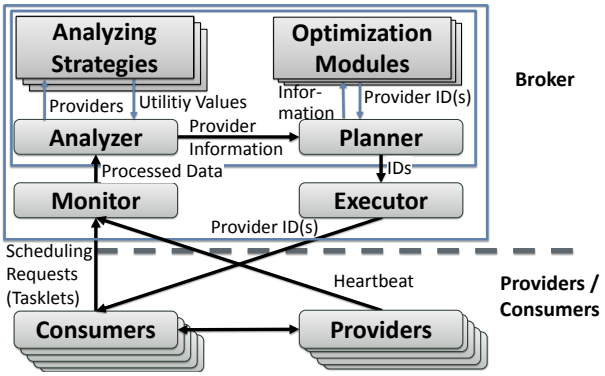

Fig. 3: System model of the scheduler exemplified with the Tasklet system. Further components of the broker are omitted.

Implicit Leave: Providers might leave the system without prior warning, e.g., when the device crashes or loses its connection to the network. Latter is often the case for mobile devices. The consumer has to detect the absence on its own.

Tasklet Abort: Even though the provider does not leave the system, it might terminate the execution of the Tasklet prematurely. This can happen for two reasons. First, the device owner stops the execution manually. Second, the Tasklet middleware withdraws resources from the TVMs since the resources are needed for local processes.

C. Fault-Awareness, Fault-Tolerance, and Fault-Avoidance

The failure model above describes the failures that can happen during Tasklet execution. To recognize these failures, the system has to be *fault-aware*. The Tasklet systems implements this fault-awareness by a heartbeat mechanism. When a provider executes a Tasklet for a consumer, it periodically informs the consumer that the execution is ongoing. The consumer detects a fault either by absent heartbeats or by an explicit message from the provider that the execution was canceled. In [19], we show how developers can use QoC to enforce a guaranteed execution. The Tasklet system implements *fault-tolerance* by re-initiating the execution of Tasklets once a drop has been detected. However, this strategy involves a delay for the consumer as the scheduling process has to be repeated and the progress of the aborted execution is lost. The optimal scheduling strategy attempts to avoid the failures described in the model above. The strategies that we present in Section V implement *fault-avoidance* by a context-aware selection of providers.

IV. Design of a Fault-Avoidant Scheduler

To implement context-awareness in the Tasklet system, we propose the design of a context-aware scheduler in this section. The scheduler is integrated into the Tasklet broker and acts as an intermediate between resource providers and consumers. This section presents the design of the scheduler component and its integration into the broker. The scheduler is a generic module and not limited to the Tasklet system. By following a modular approach, developers can easily exchange specific parts of the scheduler that are application-specific, e.g., the reception and pre-processing of context data. Figure 3 presents the system model of the scheduler and its integration into the Tasklet broker.

The scheduler integrates different context data for finding a suitable provider. Besides the task's characteristics, feedback from the providers and consumers is collected and the scheduler uses this context data for scheduling the task. This offers adaptive decision making for fulfilling a request. As common within the domain of (self-)adaptive software, our adaptive scheduler integrates a feedback structure [20]. As feedback loop structure we use the MAPE loop [21]. This loop integrates functionalities for (i) monitoring the environment, (ii) analyzing the monitored data for the need of adaptation, (iii) planning the adaptation, and (iv) executing the adaptation, i.e., changing parameters or structure [22].

Consequently, the scheduler integrates different elements for the different MAPE activities. The monitoring component is responsible for collecting data about providers, consumers, and tasks. Within the Tasklet system, the broker offers monitoring of the providers. Therefore, it captures the information of regular heartbeats that are sent from providers as well as information on finished, aborted, and timed-out Tasklets sent from consumers. Further, task requests (Tasklets) incorporate context information which is used for finding the best provider, e.g., the dimensions presented in Section II. The monitoring component offers access to all collected data.

The analyzer of the scheduler uses this information. Analyzing is based on utility functions, i.e., the analyzer evaluates the suitability of a provider by assigning a utility value to it. Then, the planner chooses the best suitable provider depending on the utility values and additional information, e.g., a Tasklet's (in general: a task's) characteristics. The split of choosing provider(s) into analyzing and planning enables an optimized decision based on the task's characteristics. The analyzing function can evaluate all providers based on their individual performance, e.g., successfully processed Tasklets in the past. Afterwards, the planning function can choose one or various provider(s) depending on the context of a Tasklet, e.g., optimize the decision for a complex Tasklet by choosing

multiple providers for parallelizing the Tasklet's execution. Following the Strategy Pattern [23], we encapsulate the algorithms for evaluating the current providers (analyzing) and deciding which provider(s) should be used (planning). This offers exchangebility of the algorithms. By using frameworks for self-adaptive system development (e.g., FESAS [24]) for implementing the scheduler, the exchange of algorithms or parameters is possible at runtime. The encapsulation of the application-specific algorithms makes the analyzing and planning components generic usable. Developers only have to specify new algorithms or reuse existing ones.

V. Fault-Avoidant Scheduling Strategies

The last section presented the design of a scheduler for fault-avoidant scheduling of tasks. The scheduler follows the MAPE principle. The core components are the analyzer for evaluating the available providers as well as the planner for choosing suitable providers. This section focuses on the implementation of scheduling strategies within these components. After a presentation of the relevant context dimensions for the implementation, we present the implementation of monitoring as well as specific algorithms for utility-based analyzing and planning.

A. Relevant Context Dimensions

The optimal scheduling strategy minimizes the number of Tasklet drops by avoiding providers that are likely to fail during Tasklet execution. As providers in the system might fail at any time, the scheduler has to predict their behavior. In Section II we presented various context dimensions that are relevant for scheduling. In the following, we focus on: (i) the residence time of providers in the system (stability) and (ii) the ability of providers to successfully execute Tasklets (reliability).

Stability: Devices in the Tasklet system show a different behavior in the time they remain connected. While stable cloud resources are available most of the time, user-owned edge devices join and leave the system frequently. We assume that the residence time, or *stability*, for each device is normally distributed, where each provider has a certain mean (μ) and variance (σ^2). The mean increases with the time that a device remains in the system on average. The variance increases when the residence time of a device in the system varies a lot. Thus, a provider that shows a high variance is less predictable than a provider with a low variance. Stable providers show a high value for μ and a small value for σ^2. Figure 4 shows an exemplified distribution of the stability for two providers.

Reliability: Besides leaving the system, providers can also abort the execution of Tasklets while remaining connected. A provider that drops Tasklets frequently is considered less effective than a provider that execute all Tasklets successfully. Within the Tasklet system, we define *reliability* as the performance of a provider to successfully execute Tasklets. For each successfully executed Tasklet, the reliability value λ of the provider is increased. When a Tasklet is dropped, the reliability is reduced. The reliability value λ is calculated by the monitoring function of the scheduler.

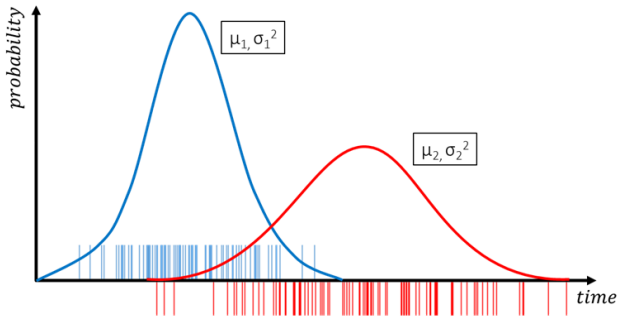

Fig. 4: Mean and Variance for two exemplary providers. The values are estimated by the monitoring functionality of the scheduler using historical information on connection times and provided for analyzing purposes. The vertical bars show show the measurement points for the respective provider.

B. Monitoring Providers

The monitoring functionality is responsible for capturing the relevant context information for scheduling and offering these information to the analyzing procedure. For measuring stability, in the Tasklet system, providers register at the brokers once they enter the system. They periodically send heartbeats to the broker to show that they are alive. Thus, the broker can measure the residence time of providers in the system. For each provider, the monitor estimates the parameters of its normal distribution based on the observations ($x_{p,i}$), where $x_{p,i}$ is the i-th residence time of provider p in the system. μ_p and σ_p^2 are estimated as follows:

$$\widehat{\mu}_p = \overline{x}_p \equiv \frac{1}{n}\sum_{i=1}^{n}(x_{p,i}) \quad , \quad \widehat{\sigma}_p^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_{p,i} - \overline{x}_p)^2$$

To learn about the reliability, the broker requires knowledge about the outcome of a Tasklet execution. Since providers will not be able to provide this feedback in case they have left the system during execution, the consumer informs the broker about the success of each execution. We implemented three approaches to calculate λ_p for each provider p:

a) Historic: λ_p is expressed as ratio of successfully executed Tasklets and all started Tasklets over time. However, this approach becomes unresponsive to changes in the behavior of a provider when the number of observations is high. A provider that has performed well for a long time but starts to perform poorly, will retain a high λ for a while.

b) Linear: For each successfully executed Tasklet, λ_p is increased by 0.01, or decreased respectively. λ_p is capped between 0 and 1. This approach reacts faster to changes than the historical calculation. However, it treats successful and unsuccessful results similarly, which might not account sufficiently for the damage that an unsuccessful execution causes.

c) Pessimistic: Each successful execution increases λ_p by 0.01. Each unsuccessful execution decreases λ_p by 0.1. Again, λ_p is capped between 0 and 1. Thus, unsuccessful executions have a larger impact on the reliability of the provider.

C. Utility-based Analyzing

We offer a generic component for utility-based analyzing that can be plugged with different algorithms for calculating the utility of providers. In principle, the analyzer receives a list with information about all providers and the Tasklet characteristics from the monitor. It evaluates each component and returns a list with utility values for them (see Algorithm 1). In this section, we present three algorithms for calculating the utility of the providers: (i) reputation-aware analysis, (ii) system-aware analysis, and (iii) runtime-aware analysis. Further analysis algorithms can be added by implementing the `calculateUtility(List<MonitoringData>, Tasklet)` function.

Algorithm 1 Analyzing Algorithm

1: **procedure** EVALUATEPROVIDERS($monitorDataList, taskContext$) $analyzingInfoList$
2: **for all** $providerData \in monitoringDataList$ **do**
3: $utility \leftarrow calculateUtility(providerData, taskContext)$
4: $analyzingInfoList \leftarrow (providerData, utility)$
5: **end for**
6: **return** $analyzingInfoList$
7: **end procedure**

1) Reputation-Aware Utility Function: We have developed a prediction algorithm that is based on the *reputation* of the providers. Azzedin et al. define reputation of a device as the '*[..] expectation of its behavior [...] within a specific context at a given time.*' [25]. In the context of the Tasklet system, the expected behavior of providers is to remain in the system and to execute Tasklets correctly. The prediction algorithm calculates a value that indicates the probability that a provider would leave the system or drop the Tasklet during execution. The algorithm has three input factors: the stability values (μ_p and σ_p^2) and the reliability value λ_p.

We normalize the values for μ_p and σ_p^2 so that all three measures fall between 0 and 1. A high mean and effectiveness result in a high reputation value. A high variance results in a low reputation value. Thus, we calculate the utility for a provider as an average of the mean, the effectiveness, and the corrected variance.

$$U_p = \frac{||\mu_p|| + (1 - ||\sigma_p^2||) + \lambda_p}{3} \tag{1}$$

2) System-Aware Utility Function: Taking the stability and the reliability of providers into account, we can improve the quality of scheduling decisions. However, this approach has some limitations. First, in this model, the three parameters to compute the utility all have the same weight. This might fit for some environments but will not be optimal for all provider pools. Second, the dynamics of providers entering and leaving the system might change over time as well as their reliability. Depending on the composition of providers, the optimal scheduling strategy, and thus the calculation of the utility, might change over time. Thus, we extended the reputation-aware scheduling strategy by two features: First, we assign weights α, β, and γ to the parameters of the utility function. Second, we made these weights adaptable, based on the current context of the system. In a system with high dynamism, the stability parameters μ_p and σ_p^2 are emphasized. In a system with many corrupt or unreliable providers, γ is increased for increasing the significance of λ_p at the cost of μ_p and σ_p^2. Thus, the adapted equation to compute the utility value looks as follows:

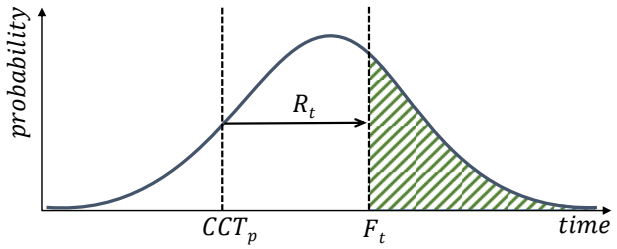

Fig. 5: Computation of the probability that a provider would remain in the system until the execution of the Tasklet has finished (F_t). This probability is used to compute the utility of the runtime-aware strategy. (CCT_p = current connection time of the provider, R_t = runtime of the Tasklet)

$$U_p = \alpha * ||\mu_p|| + \beta * (1 - ||\sigma_p^2||) + \gamma * \lambda_p \tag{2}$$

3) Runtime-Aware Utility Function: So far, we have considered context dimensions of providers and the system itself neglecting the characteristics of the Tasklet. However, when a Tasklet is dropped during execution, the current progress of the execution is lost and the Tasklet has to be rescheduled. Tasklets with a longer runtime suffer more from a drop than short Tasklets. Thus, the scheduler should assign providers with the highest utility to long running Tasklets to reduce the probability that the execution of a long running Tasklet is canceled. With more knowledge about the runtime of a Tasklet, we can estimate how likely it is that a consumer would remain long enough in the system to execute the Tasklet completely. For this approach, we need to know the runtime of Tasklets. In general, this runtime is not known but have to be estimated by either learning from historic data or by code analysis. Estimation methods for execution times have been discussed in [26]. In our model, we assume that the runtime of Tasklets R_t is estimated by the consumer and provided to the scheduler.

We compute the utility of providers by a linear combination of two parameters: 1) the probability of a successful execution and 2) the reliability of the provider. Whereas the reliability is computed as in the utility functions above, the probability of a successful execution depends on the estimated stability of providers and the runtime of the Tasklet that is to be executed. Using the current connection time (CCT_p) and R_t, we predict the point of time F_t when the execution of the Tasklet would be finished. Using CCT_p, the two parameters μ and σ^2, and the assumption that the connection time of providers is distributed normally, we calculate the probability that a provider is still active at F_t. Figure 5 illustrates how we compute this probability. The likelihood ($\tau_{p,t}$) that a provider

remains long enough in the system to execute the Tasklet successfully can be computed as follows:

$$\tau_{p,t} = \Phi(1 - (CCT_p + R_t))$$

where $\Phi(x)$ is computed as:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}t^2} dt \ , \ F(x) = \Phi\left(\frac{x-\mu}{\sigma}\right)$$

As a result, the overall utility U can be computed as a linear combination of the likelihood τ and the reliability λ:

$$U_{p,t} = \epsilon * \tau_{p,t} + \gamma * \lambda_p \tag{3}$$

Compared to the previous approaches, the utility is not the same for all Tasklet, but depends on the runtime of Tasklets. Thus, it has to be computed for each Tasklet individually.

D. Planning

The planner performs the actual scheduling decision. It receives a list of providers with their corresponding utility values from the analyzer. Using this list, the planner returns a suitable set of providers depending on the specified planning algorithm and the characteristics of the Tasklet. We have implemented four different planning strategies that can be combined with any analyzing module: (i) naive, (ii) active-aware, (iii) idle-aware, and (iv) greedy. However, further planning algorithms may be added by implementing the interface `ISchedulingPlanningAlgorithm`.

Naive: The naive planner randomly selects providers from the list of known providers. It does not take any further context information into account.

Active-Aware: To avoid delays caused by inactive providers, the second planning approach randomly selects active providers from the list provided by the analyzer. This planning approach requires knowledge about the current state of providers which is monitored using heartbeats.

Idle-Aware: Providers that are considered in the scheduling decision might already be busy running Tasklets or their resources are used locally. Thus, this planning approach considers the current workload of active providers and choose idle planners only. Load information can be piggybacked on heartbeats from the providers.

Greedy: The most complex planning strategy selects the idle providers with the highest utility value to increase the probability that a Tasklet is executed successfully. As a result, providers with a high utility value get selected more frequently and the overall performance is expected to increase.

VI. Evaluation

We ran the evaluation of the scheduling strategies in a simulated distributed environment. While evaluation results from the Tasklet system in a real-world testbed with more than 160 physical devices and cloud instances can be found in [19], we decided to simulate the environment for the evaluation of the scheduler for the following reasons: First, to reliably measure the performance of the scheduler, the system needs to span a large amount of resources. In real-world testbeds, it is extremely costly to set up hundreds or thousands of devices. Second, in the evaluation we simulate node failures, Tasklet drops, and several context states to demonstrate how the scheduler reacts to different system conditions. Controlling these parameters in a real-world testbed is hardly feasible.

A. Simulation Model

The simulation models consists of the three entities of the Tasklet system: providers, consumers, and brokers. For this evaluation we use a centralized pattern with a single broker that handles all resource requests from the brokers. There are 1000 providers that randomly enter and leave the system randomly based on their underlying probability distribution and drop a Tasklet with the probability δ_p. 100 consumers offload Tasklets with different runtimes to the brokers in random intervals.

B. Performance Measure

To measure the performance of the scheduling decisions, we compute the costs for consumers. Consumers attempt to minimize the costs of executing Tasklets, where costs can be defined in terms of time, energy, or money. In our model, we do not consider energy consumption or money, but focus on the execution time as a performance measure for consumers. We define the term *completion time* (T_{comp}) as the time that a Tasklet requires to be executed successfully. The completion time includes the time that is required for scheduling the Tasklet to a provider (T_{sch}), the execution time on providers (T_{ex}), the time required for rescheduling (T_{re}), and the time required for returning the result to the consumer (T_{res}). T_{re} includes the delay until the consumer has detected the Tasklet drop and the time for scheduling the Tasklet to another provider. Since the Tasklet can fail several times, there can be multiple execution and rescheduling intervals. Thus, the optimal scheduling strategy can be defined as

$$min(T_{comp}) = min(T_{sch} + \sum_{i=1}^{n+1}(T_{ex}) + \sum_{i=1}^{n}(T_{re}) + T_{res})$$

where n is the number of failures, respectively the number or rescheduling attempts. In [19], we have shown that we can decrease (T_{ex}) by scheduling Tasklets to powerful devices. In this paper, we ignore the hardware heterogeneity of resources and assume that all devices execute Tasklets with the same speed. Instead, we try to reduce n (the number of faults, respectively the number of required retransmissions) to minimize (T_{comp}).

C. Baseline

We measure two simple scheduling algorithms as baselines for the evaluation. These measures set the benchmark for more complex strategies that we have presented in the previous section. We have implemented two basic scheduling approaches that select brokers on a random basis, without any quantitative analysis.

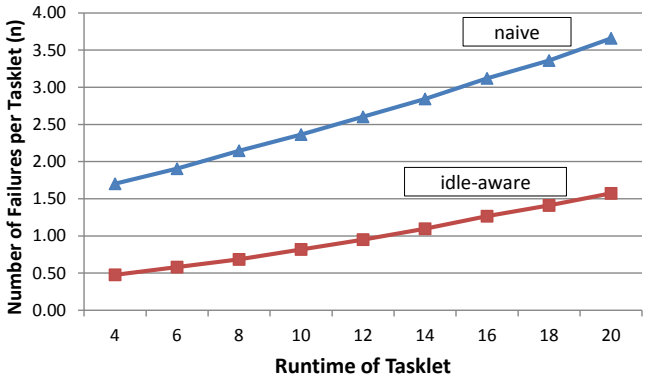

Fig. 6: Baseline measures for the simple scheduling mechanism in the Tasklet system. The naive baseline has no context information at all. The idle-aware strategy has knowledge about whether the provider is active or idle.

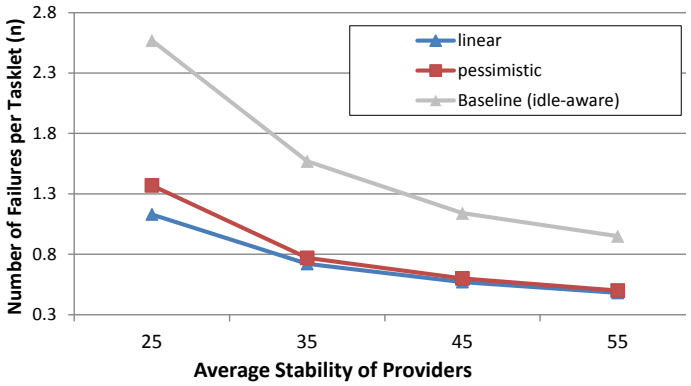

Fig. 7: Reputation-aware scheduling for different system environments (highly dynamic to less dynamic). Both strategies (linear and pessimistic) clearly outperform the baseline and further reduce the number of failures.

The *naive* baseline represents a best-effort scheduling approach that does not make use of any context information at all. It randomly matches a Tasklet with a provider from its resource pool and forwards these information to a consumer. This provider might have left the system or is currently busy executing other Tasklets. As a result, it might not execute the Tasklet and the consumer has to request a new provider.

The *idle-aware* baseline uses the context information whether the provider is still active and whether it is idle or not. While this approach requires some context-awareness it reduces the number of unsuccessful Tasklet requests from consumers to inactive or busy providers.

Figure 6 shows the results of the baseline evaluation. The plots indicate the average number of failures per Tasklet, sorted by the execution time of Tasklets. The results provide two insights into the system: First, implementing some context-awareness already increases the quality of scheduling decisions significantly. By monitoring the state of the providers, many retransmissions can be avoided. Second, Tasklets with a higher execution time are more likely to fail and have to be re-initiated more frequently. This behavior is expected as the probability that a provider leaves during Tasklet execution increases with the execution time.

D. Reputation-Aware Strategy

The baseline evaluation has shown that context information can improve scheduling decisions. In the next step, we evaluate the reputation-aware strategy that selects the provider based on the utility that is computed according to Equation 1. To adapt the reliability parameter λ, we used the linear as well as the pessimistic approach (see section V-B). We simulated four different system environments, ranging from a highly dynamic one (with a mean residence time of providers of 25) to a more stable environment with a mean residence time of 55. The results in Figure 7 show that the reputation-aware strategy clearly outperforms the idle-aware baseline which is shown as a reference. the failure rate decreases in less dynamic environments. Further, the linear adaption for the reliability (λ_p) performs better than the pessimistic strategy.

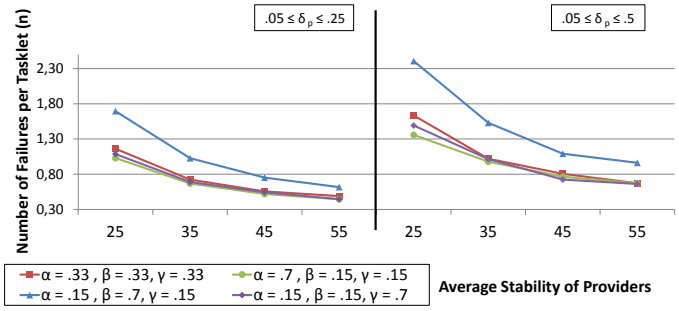

Fig. 8: System-aware scheduling for different environments. The plots show scheduling results from different combinations of weights in Equation 2. The tests have been performed with two different Tasklet drop rates δ_p.

E. System-Aware Strategy

As the system characteristics might change over time, the scheduler should be able to adapt to the new situation. In the system-aware strategy, the scheduler monitors the system context and adapts the weighting parameters α, β, and γ (see Equation 2), if necessary. Again, we ran the system-aware scheduler in the same four system environments as in the previous test. In each environment, we tested different combinations of the weighting parameters. The results in Figure 8 show that the algorithm performs best if the highest weight is put on the expected residence time. This becomes even more relevant when the system is very dynamic ($stability = 25$) and providers join and leave frequently. Putting to much weight on the variance, the performance of the scheduler decreases significantly. The system-aware strategy outperforms the baseline (not shown here) and slightly improves the result from the reputation-aware strategy.

The drop rate δ_p defines the ratio to which a provider drops a Tasklet during execution. So far, each provider got assigned a δ_p between 0.05 and 0.25. For the system-aware scheduling, we also test a second scenario with a δ_p between 0.05 and 0.5. As a result, the average number of failures per Tasklet increases. The selection of the optimal weights remains the same.

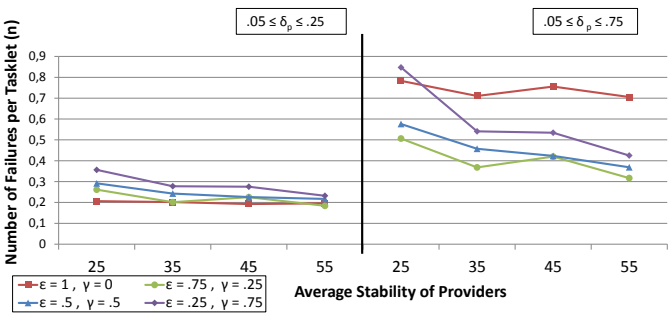

Fig. 9: Runtime-aware scheduling systems in four different environments. In an environment with low drop rates δ_p, the strategy $\epsilon = 1, \gamma = 0$ is optimal. For less predictable environments ($.05 \leq \delta_p \leq .75$), the parameters have to be adjusted.

F. Runtime-Aware Strategy

In the last step, we evaluated the runtime-aware strategy. It does not only take the context of providers and the system into account, but also uses the context of the consumer, as it considers the runtime of a Tasklet for the scheduling decision. Similar to the two approaches before, we tested the runtime-aware strategy in four environments with different average residence times of the providers. We used different weights on the parameters ϵ and γ in Equation 3. The results in Figure 5 show that the best result can be achieved when the full weight of the function is put on ϵ, and thus the scheduling decision only depends on the likelihood of a provider to remain long enough in the system. This mechanism works that well that it does not decrease in performance when the system becomes more dynamic.

However, when we increase the drop rates δ_p to values between 0.05 and 0.75, this strategy does not perform well anymore. Instead, in highly unreliable environments, it is beneficial to take the reliability measure λ_p into account and put some weight on the factor γ. It is the responsibility of the scheduler, to monitor the environment and to adapt the scheduling parameters accordingly.

VII. Related Work

Since the advent of distributed computing, a lot of research has been conducted in the field of fault-tolerant scheduling. The approaches can be categorized into fault-aware and fault-avoidance strategies. In [27], Avizienis et al. provide a taxonomy of fault-tolerance and fault forecasting.

1) Fault-Aware Strategies: A survey on fault-tolerance mechanisms in grid computing can be found in [28]. [10] uses replication of tasks to ensure a certain fault-tolerance level in mobile grid environments. They do not guarantee a reliable execution but increase the probability of success. [9] implements fault recovery where failed nodes will either be replaced or repaired. To increase the performance of fault discovery, the system introduces constraints on the executed tasks. In [29], a fault-aware scheduling for desktop grid systems is presented. The authors implemented eight fault-aware policies to increase the scheduling performance. Tang et al. [30] introduce a reliability-driven scheduling architecture to deal with node failures in heterogeneous environments.

2) Fault-Avoidance Strategies: Rood et al. [11] provide fault-avoidance by availability prediction and replication. Their prediction is based on historical data about job completions. Lee et al. [12] predict the availability of mobile devices based on users' mobility patterns. Based on these patterns they classify devices into three categories of availability. Ren et al. [14] use a semi-Markov process to predict the availability of resources. They monitor the free CPU load and free memory to detect state changes. In the approach of Chakravorty et al. [31], the authors assume that failures can be predicted and a running task can be migrated before a fault occurs. Duan et al. [32] use clustering and learning algorithms on workflow and historic data of resources to predict faults. Kang et al. [33] predict failures based on observed inter-arrival times of failures in managed machines. Sonnek et al. [34] and Damiani et al. [35] propose reputation-based scheduling approaches for peer-to-peer systems.

Besides fault-awareness, there are other scheduling approaches that take one or more context dimensions into account. Sparrow [36] implements a decentralized scheduling approach to maximize the throughput in a distributed computing system. [37] introduces a hybrid scheduling algorithm that takes different kinds of resources into account. For each type of resource, an appropriate scheduling mechanism is used. Stork [38] presents an efficient data-aware scheduling in grid computing. Scheduling decisions are based on task-specific and global data policies. Xie et al. [39] discuss a security-aware scheduling strategy for real-time applications.

VIII. Conclusion

In this paper, we introduced an architecture for context-aware and fault-avoidant scheduling in heterogeneous distributed computing systems. We proposed three analyzing and four planning algorithms for incorporating context dimensions into the process for distributed real-time scheduling. We have implemented the scheduler in our Tasklet system and evaluated the proposed analyzing and planning algorithms. The results show that taking the context of consumers, providers, and the system into account, can tremendously increase the performance of scheduling decisions. The performance increase does not come for free. Monitoring the context, analyzing the data, and selecting the optimal provider requires additional overhead that increases with the number of context dimensions.

For future work, we plan to implement machine learning mechanisms to make the system self-adaptive and learn how to optimally adjust weights and parameters. Further, we plan to verify the assumption that the stability of the providers is normally distributed. Therefore, we plan to run a real-world study an monitor the behavior of the resource providers.

IX. Acknowledgements

This work was supported by the German Research Foundation (DFG) and the Julius Paul Stiegler Memorial Foundation. We thank Martin Breitbach for his valuable contributions.

REFERENCES

[1] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: Toward Distraction-Free Pervasive Computing," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22–31, 2002.

[2] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "Gaia: A Middleware Infrastructure for Active Spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, 2002.

[3] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "BASE - a Micro-broker-based Middleware for Pervasive Computing," in *Proceedings of the International Conference on Pervasive Computing and Communications*. IEEE, 2003, pp. 443–451.

[4] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - A Component System for Pervasive Computing," in *Proceedings of the International Conference on Pervasive Computing and Communications*. IEEE, 2004, pp. 67–76.

[5] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, "Evaluation of job-scheduling strategies for grid computing," in *Proceedings of the International Workshop on Grid Computing*. Springer, 2000, pp. 191–202.

[6] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software: Practice and Experience*, vol. 32, no. 2, pp. 135–164, 2002.

[7] F. Xhafa and A. Abraham, "Computational models and heuristic methods for grid scheduling problems," *Future generation computer systems*, vol. 26, no. 4, pp. 608–621, 2010.

[8] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[9] S. Guo, H.-z. Huang, Z. Wang, and M. Xie, "Grid Service Reliability Modeling and Optimal Task Scheduling Considering Fault Recovery," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 263–274, 2011.

[10] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou, "Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments," *Future Generation Computer Systems*, vol. 23, pp. 163–178, 2007.

[11] B. Rood and M. J. Lewis, "Availability Prediction Based Replication Strategies for Grid Environments," in *Proceedings of the International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 25–33.

[12] J. Lee, S. Song, J. Gil, K. Chung, T. Suh, and H. Yu, "Balanced Scheduling Algorithm Considering Availability in Mobile Grid," in *Proceedings of the International Conference on Grid and Pervasive Computing*. Springer, 2009, pp. 211–222.

[13] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 253–285, 1997.

[14] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi, "Resource Failure Prediction in Fine-Grained Cycle Sharing Systems," in *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, 2006, pp. 19–23.

[15] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *Proceedings of the International Symposium on Handheld and Ubiquitous Computing*. Springer, 1999, pp. 304–307.

[16] Amazon.com, Inc., "Amazon EC2 Spot Instances," http://aws.amazon.com/de/ec2/spot, accessed: 2016-06-24.

[17] A. Schmidt, M. Beigl, and H.-W. Gellersen, "There is more to context than location," *Computers & Graphics*, vol. 23, no. 6, pp. 893–901, 1999.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM, 2010, pp. 49–62.

[19] D. Schafer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: "Better than Best-Effort" Computing," in *Proccedings of the International Conference on Computer Communication and Networks*. IEEE, 2016, pp. 1–11.

[20] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5525, pp. 48–70.

[21] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[22] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[24] C. Krupitzer, F. M. Roth, S. VanSyckel, and C. Becker, "Towards Reusability in Autonomic Computing," in *Proceedings of the International Conference on Autonomic Computing*. IEEE, 2015, pp. 115–120.

[25] F. Azzedin and M. Maheswaran, "Integrating trust into grid resource management systems," in *Proceedings of the International Conference on Parallel Processing*. IEEE, 2002, pp. 47–54.

[26] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, p. 36, 2008.

[27] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[28] R. Garg and A. K. Singh, "Fault Tolerance in Grid Computing: State of the Art and Open Issues," *International Journal of Computer Science & Engineering Survey*, vol. 2, no. 1, 2011.

[29] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski, "Fault-aware scheduling for bag-of-tasks applications on desktop grids," in *Proceedings of the International Conference on Grid Computing*. IEEE, 2006, pp. 56–63.

[30] X. Tang, K. Li, R. Li, and B. Veeravalli, "Reliability-aware scheduling strategy for heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 941–952, 2010.

[31] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive Fault Tolerance in MPI Applications Via Task Migration," in *Proceedings of the International Conference on High Performance Computing*, 2006, pp. 485–496.

[32] R. Duan, R. Prodan, and T. Fahringer, "Short Paper : Data Mining-based Fault Prediction and Detection on the Grid," in *Proceedings of the IEEE International Conference on High Performance Distributed Computing*, 2006, pp. 305–308.

[33] W. Kang and A. Grimshaw, "Failure Prediction in Computational Grids," in *Proceedings of the Annual Simulation Symposium*, 2007, pp. 275–282.

[34] J. Sonnek, A. Chandra, and J. Weissman, "Adaptive reputation-based scheduling on unreliable distributed infrastructures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 11, pp. 1551–1564, 2007.

[35] E. Damiani, D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante, "A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks," in *Proceedings of the conference on Computer and communications security*. ACM, 2002, pp. 207–216.

[36] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.

[37] M.-A. Vasile, F. Pop, R.-I. Tutueanu, V. Cristea, and J. Kołodziej, "Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing," *Future Generation Computer Systems*, vol. 51, pp. 61–71, 2015.

[38] T. Kosar and M. Balman, "A new paradigm: Data-aware scheduling in grid computing," *Future Generation Computer Systems*, vol. 25, no. 4, pp. 406–413, 2009.

[39] T. Xie, X. Qin, and A. Sung, "Sarec: A security-aware scheduling strategy for real-time applications on clusters," in *Proceedings of the International Conference on Parallel Processing*. IEEE, 2005, pp. 5–12.