Context-Aware Data and Task Placement in Edge Computing Environments

Martin Breitbach, Dominik Schäfer, Janick Edinger, Christian Becker
University of Mannheim
Mannheim, Germany
{martin.breitbach, dominik.schaefer, janick.edinger, christian.becker}@uni-mannheim.de

***Abstract*—Computationally intensive tasks of IoT applications can be offloaded to powerful devices in the edge. Code offloading reduces energy consumption and increases performance. However, applications that use face recognition, machine learning, or image rendering, rely on large amounts of data. The transfer of this data leads to latencies which contradicts the responsiveness required by many pervasive applications. As a solution, decoupling the data from the tasks allows to apply new scheduling strategies that place data on remote devices before the actual task execution. Grid computing approaches use this technique effectively, however, edge computing introduces further challenges such as device fluctuation and heterogeneity.**

In this paper, we propose a data management approach for edge computing environments that decouples data placement from task scheduling. We present a multi-level scheduler, which places data on resource providers in the system considering multiple context dimensions. The scheduler allocates tasks according to the current context and observes the state during runtime. If required, the system adjusts the number of data copies to optimize the trade-off between execution latencies and data management overhead. The paper has three contributions: (1) a context-aware multi-level scheduler, (2) the integration of four data placement, three task scheduling, and three runtime adaptation algorithms, (3) an evaluation in a real-world testbed.

***Index Terms*—Data placement, task allocation, middleware, edge computing, distributed computing**

I. Introduction

Offloading computation in grid and edge environments has several benefits regarding execution performance, energy consumption, and utilization of excess capacities [1]–[3]. For a remote execution, systems generate closed entities of computation, which consist of code, parameters, and data [4]. These tasks are shipped from a resource consumer, which runs the application, to resource providers. The providers execute the tasks and return results to the consumer. When tasks require large chunks of data, scheduling becomes more crucial due to data transfer times, storage restrictions, and varying provider reliability. As a solution, the separation of data from the closed computation entity offers new scheduling potential. If data enters the system before the respective tasks, the system can already place the data on resource providers. The transmission of the actual task is then minimized to code and parameters.

The independent scheduling of data and tasks poses several challenges such as workflow planning, optimal storage utilization, and API support for developers. As far as data and task placement is concerned, two questions arise in particular: (1) how many data replicas should the environment store in total and (2) which resource providers are most suitable to store these replicas? The number of data replicas can range from no replicas at all to a full replication where each provider in the nearby environment maintains a local replica. The overhead for a full replication in terms of data distribution, occupied storage, and maintenance is large. However, this replication strategy is most promising from a task execution perspective since turnaround times do not contain any data transfer latencies. On the other side, no replication or a single replication is easy to create and maintain, but may lead to high execution latencies. Between these two extremes, n-replication strategies exist. These strategies can balance the trade-off between data transfer overhead and execution delay as they allow to adjust the number of replicas to the system context.

Data management is in the focus of distributed computing researchers for decades. In cluster and grid computing, optimized data placement leads to reduced execution times [5]–[7]. However, the placement strategies are rather static due to the stability of grid environments. With the emergence of cloud computing, the problem moved in another direction [8]–[11]. In cloud computing, the environment is mostly homogeneous. Edge computing environments now introduce new challenges [12]. Devices are user-controlled and may leave the system at any time. Further, the landscape of edge environments is heterogeneous. Hence, fluctuation and heterogeneity increase the complexity for data placement strategies.

In this paper, we propose a data management system that copes with the characteristics of the edge. It decouples data and task scheduling and accomplishes resource allocation on three levels: First, before the actual runtime of tasks, the *data placement level* places data on providers considering the system's context. Second, the *task scheduling level* allocates tasks on the most suitable providers. Third, the *runtime adaptation level* monitors the quality level of task execution and adapts data placement in a control loop if necessary. As a result, our approach optimizes the trade-off between execution latencies and data overhead based on the current context.

After the related work section, the remainder of this paper contains our three contributions: First, a multi-level scheduling architecture. Second, we present data and task allocation algorithms that reduce task execution latencies while minimizing the data overhead in edge environments. Third, we evaluate our approach in a real-world testbed with three different applications. Finally, we conclude our work in the last section.

978-1-5386-9148-9/19/$31.00 ©2019 IEEE

II. Related Work

In distributed computing, data management and the dependencies to task scheduling have always been a major research focus. Our approach is related to prior research on *task scheduling* and *replication*. Several approaches also offer a *combination of data and task scheduling* similar to this paper.

1) Task scheduling: Early grid computing research has already considered the impact of data transfer times for task scheduling on remote providers [5]. Braun *et al.* [13] compare 11 heuristic algorithms for task scheduling in grids. The majority of these algorithms considers data transfer times. A genetic algorithm performed best in all scenarios. Several related approaches such as [6], [7], [14]–[16] developed more sophisticated grid scheduling strategies. In [17], McClatchey *et al.* introduce *DIANA*, a context-aware scheduling strategy that allocates tasks minimizing the network, computation, and data transfer cost. Liu *et al.* [18] and Taheri *et al.* [19] use nature-inspired algorithms to schedule workflows in the grid. More recently, cloud computing research drew new attention to scheduling strategies that consider data location [10], [11]. Casas *et al.* [20] present a scheduler for cloud computing that allocates scientific workflows by exploiting data reuse and replication strategies. Li *et al.* [21] propose a scheduling strategy for batch jobs in MapReduce that considers the data locality of tasks. In the context of edge computing, Elbamby *et al.* [22] minimize computing latency via joint task offloading and proactive caching of popular tasks. They use computing as well as storage resources and consider the location of edge nodes for the offloading decision. Although data location and data transfer times are an important part of the aforementioned scheduling approaches, none of them considers moving the data to further improve system performance.

2) Replication: Especially in the context of grid computing, several approaches use replication to optimize task execution times [23]–[25]. Chervenak *et al.* [26] combine *HTCondor*'s [27] workflow management with a data placement service. Since 2011, HTCondor integrates the *Stork* [28] data scheduler that focuses on the matchmaking of code and data on the provider side. However, it neither decides on the location of the data nor performs any data placement before task runtime. In [29], Nukarapu *et al.* present centralized and decentralized versions of a greedy replication strategy that minimizes the total delay to access input files for a given set of tasks.

3) Combining data placement and task scheduling: Similar to our approach, several systems use both, strategic data placement and task scheduling, to accelerate execution times. Ranganathan and Foster understand data and task allocation as two independent processes [30]. Scheduling tasks on providers that already store the input data in combination with replication performed best in their evaluation. Desprez and Vernois combine data management and scheduling with a steady-state approach [31]. Assuming that data and tasks are known in advance, their algorithm computes the optimal data distribution. Tang *et al.* introduce an approach that periodically updates data placement based on the usage in the past [32]. In case of a task request, it schedules the task on the resource with the shortest expected turnaround time. Chakrabarti and Sengupta organize providers in virtual clusters and optimize the current data distribution according to demand, frequency of data access, and expected latency [33]. These approaches focus on rather homogeneous environments and do not cover the characteristics of edge computing systems such as fluctuation. Cameron *et al.* present three heuristics to determine whether the provider should keep a replica of the input data of a recently-finished task. Chang *et al.* also first schedule tasks on suitable providers and then decide to store the input data permanently or to delete it based on a *least frequency used* (LFU) heuristic [34]. Both approaches do not optimize data distribution before task runtime. Tang *et al.* [35] propose a data-centric fog architecture that reduces data transfer overhead and offers low latencies. However, it has a fixed hierarchy and data flow. The *Nebula* architecture [36] supports distributed data-intensive applications through a close interaction between storage and compute resources in an edge cloud scenario. Another edge computing approach by Li *et al.* [37] includes an iterative task placement algorithm where edge servers forward tasks to other edge servers that have the required data. Similar to Nebula, the data placement is rather static and does not provide adaptation at runtime.

III. An Architecture for Data Management

In the following section, we first introduce the major challenges of data and task allocation in edge environments. Then, we introduce an architecture consisting of six different components that is able to cope with these challenges. Finally, we describe the design of our multi-level scheduler.

A. Challenges

Decoupling computation and data scheduling in grid environments has been proposed by Ranganathan and Foster as early as 2002 [30]. While the edge paradigm with no doubt has its merits, it introduces new challenges for data and task placement compared to grid computing. First, *heterogeneity* in terms of hardware such as computation power, network connection, or storage capabilities is typical for edge environments [38]. Second, edge devices can leave the system gracefully or without notice, leading to *fluctuation*. Whereas some devices appear at regular intervals others might enter the system sporadically or only just once. Large data files should be transferred to appropriate devices. Edge computing environments consist of a large number of devices with a high bandwidth connectivity [39]. This benefits *edge applications* that make use of these excess capacities. In contrast to batch applications, tasks in edge environments require high responsiveness. Sluggish data replication, queuing, and aborted executions lead to unacceptable delays. Task scheduling decisions cannot be made in a statical way but need to be done dynamically upon task creation. Ideally, the data placement has happened beforehand. The novel challenges of edge computing require suitable data and task placement.

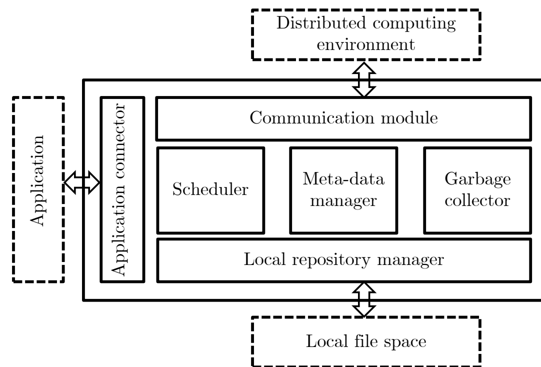


Fig. 1. The architecture of the data management system. The system runs decentralized on each resource. It communicates via the interfaces application connector, local repository manager, and communication module with applications, local file space, and distributed computing environment. The core components scheduler, meta-data manager, and garbage collector orchestrate the data management process.

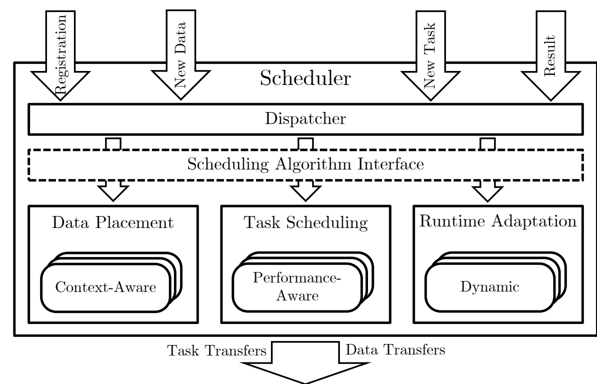


Fig. 2. Multi-level scheduler design for data placement prior to task execution, task placement, and runtime adaptation. The modular design allows to integrate various strategies on each level for particular contexts.

B. System Architecture

Our system architecture is able to cope with these circumstances. Each entity in the edge environment runs an own instance of the system. Depending on the architecture of the distributed computing system that uses the data management, some components may run on central instances such as brokers only. This instance consists of six components that interact with each other. Three of these components are interfaces. The *Application Connector* communicates with the distributed computing applications. It receives new data from an application or forwards results. A second interface, the *Local Repository Manager*, provides access to the local file space. It stores data permanently, manages access to files, and executes garbage collection. To transfer data to other resources and to receive data, the *Communication Module* handles the connection to the distributed computing environment.

The three interfaces offer essential features to the core components which are responsible for orchestrating the data management process. The *scheduler* places data strategically in the system and allocates tasks to resource providers. To make informed decisions, it is crucial to have an up-to-date view on the distributed computing system. The *meta-data manager* offers this view. It aggregates information about the resources in the system, the data files that they store, and further meta-data such as latencies. Similar to the scheduler, the *garbage collector* also relies on the meta-data manager. It conducts the garbage collection process, ideally in close cooperation with the scheduler. Figure 1 shows the architecture of the data management system with six components and the interaction with three entities outside of the system.

C. Multi-Level Scheduler

In the following, we focus on the design of the scheduler of the data management system. This component is of key importance as it determines where to place data and tasks strategically in the system. Hence, it influences the performance of the overall distributed computing system substantially. We use an integrated scheduling approach that manages both, data placement and task scheduling. This integration into one scheduler leads to better optimization of the system state and less overhead for negotiation between two independent schedulers. In addition, the integrated design avoids oscillation effects, that occur if separated data and task schedulers repeatedly counteract each other's decisions. Figure 2 shows the scheduler design in detail.

The multi-level scheduler includes a *dispatcher* component that forwards incoming events to the appropriate level. If a resource enters new data into the system, the dispatcher triggers the *data placement level*. Before the actual runtime of tasks, the data placement level decides whether to replicate new data and where to place these replicas. To avoid additional assumptions, our approach does not assume any knowledge about the future workflow or dependencies between data and certain tasks at this level. In case of a new task, the dispatcher forwards the task request to the *task scheduling level*. This level allocates tasks on the most suitable devices in the system. Depending on the particular strategy and the data dependencies of the task, the allocation may exploit the data distribution created previously by the data placement level. An incoming result of a computation or the (de-) registration of a device is dispatched to the *runtime adaptation level*. As initial data placement decisions can be ineffective, this level ensures the adjustment of the data placement to the current system state at runtime. For instance, if certain data is frequently used, the placement of an additional replica may be beneficial.

The effectiveness of scheduling strategies depends highly on the context. Strategies perform differently if network churn, task complexity, or average data file size change. To consider this and to allow for extensibility, each of the scheduler's levels contains different *strategies*. The *scheduling algorithm interface* defines functions that all strategies need to provide. Now, programmers can tailor new scheduling algorithms to the respective distributed computing system and integrate them easily. Additionally, this modular design improves adaptability. A self-adaptive scheduler may choose from different scheduling algorithms at runtime based on the current context.

IV. Data-centric scheduling in edge environments

This section presents the strategies integrated into each level of the multi-level scheduler. The first level is the data placement, which happens prior to the task execution. The second level is the task scheduling level. The third level is the runtime adaptation level, which uses a control loop for self-adaptation to adjust the numbers of data replicas in the system. Thus, we develop an overall scheduling strategy that mitigates the impact of device heterogeneity in the first place and reacts to bottlenecks during runtime.

A. Data Placement Level

The data placement level optimizes the distribution of new data in the system by applying replication. Replication is the management of various copies of the same data distributed on different computers [40]. It leads to shorter turnaround times of tasks at the cost of data management. Thus, the design of a replication-based scheduling strategy needs to balance the tradeoff between fast execution and data overhead. In general, two extreme strategies exist: *no replication* and *full replication*. However, between those two extreme strategies, approaches exist which balance overhead and performance. Next, we investigate three basic data placement strategies and introduce our context-aware data placement strategy tailored to edge computing environments, all summarized in Figure 3.

1) Basic Replication Strategies: In the *no replication* strategy, the scheduler always couples data and code transfers. It does not perform any data placement before task runtime. Thus, the strategy optimizes initial data transfer overhead. However, it leads to high task execution times as each execution will always include the latency for the data transmission. In the case that data and tasks are simultaneously entered into the system, this strategy is inevitably used as fallback.

With the *1-replication* strategy, the data management system places a single replica on a random provider before task runtime. The initial data distribution costs are constant and independent from the environment. Further, for periodic execution of tasks, the replica can be reused without any data transmission during task runtime. However, the 1-replication strategy does not support concurrent executions on multiple devices nor reliability in case of fluctuation.

The *full replication* strategy replicates all data objects on all providers. If new data enters the system, data management transfers it to all resources. This strategy minimizes the task execution time. To achieve this, unnecessary data transfers will happen as not all providers will execute all tasks.

2) Context-aware Replication: Different task characteristics and system conditions influence the effectiveness of the aforementioned generic replication strategies. To cope with the dynamism of modern computing environments, we introduce *context-aware replication*. This mechanism tunes the parameters of the replication strategy based on context information gathered by the meta-data manager.

Context-aware replication encompasses two decisions for each new data object d. First, it needs to determine the number of replicas n. Second, the mechanism chooses the devices that will store the replicas. A separation of these two steps ensures a reasonable number of replicas during execution without relying on device monitoring or assuming certain device characteristics. Concerning the first decision, four context variables play a major role. The appropriate number of replicas depends on the data size, the remaining storage capacity of the system, the current fluctuation, and the application. We model the current state of these four variables with normalized coefficients ranging from 0 to 1. The coefficient C_{data} describes the relative data size by comparing the absolute data size s_d to the maximum allowed data size in the system s_{max}. In a system with k providers $p_1, ..., p_k$, the coefficient for the remaining storage capacity, C_{cap}, is the sum of the free storage c_f of all devices divided by the sum of the total storage c_t of each device.

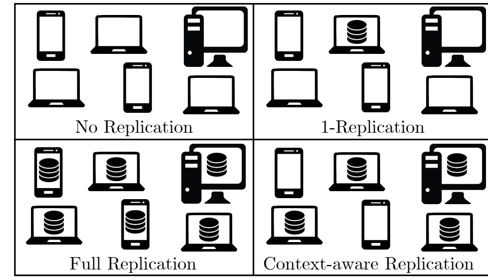


Fig. 3. The four data placement strategies that are integrated in the data placement level. No replication does not place data prior to task executions. The 1-replication strategy places a single data copy on a provider in the environment. With full replication, all resources in the system hold a data replica. The context-aware replication strategy decides on the level of replication and the most suitable providers based on context information.

The influence of the current fluctuation on the desired number of replicas is modeled in the coefficient C_{flu}. To calculate this coefficient, we use a sliding window approach and determine the mean residence times for each device that has been part of the distributed computing system in this time window. The average of these mean residence times quantifies the current provider stability $Stab_{prov}$ in the system. However, there is no linear relation between provider stability and the appropriate number of replicas. A high number of replicas is beneficial in systems with moderate stability values. In unstable systems with high fluctuation, less replicas should be chosen since replicas are likely to leave the system before any task execution can happen on the device that stores the replica. Additionally, systems with high stability values also require less replicas. Devices barely leave the system which makes having another copy of the data inefficient. To model this non-linear relation, we apply a polynomial function as shown in Figure 4. This function represents an example and can be exchanged depending on system or application characteristics.

Moreover, the application characteristics influence replication. To incorporate these application-specific characteristics into the replication decision, the coefficient C_{app} represents whether the application might require a higher number of replicas. The data placement level calculates this coefficient as the average of the data availability factor f_{ava} and the

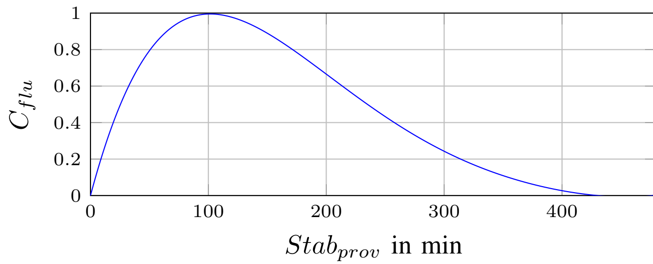


Fig. 4. Function to determine C_{flu} based on the provider stability value.

parallelism factor f_{par} of the application that entered the data into the system. Applications with a high data availability factor benefit from having multiple replicas since they require to continuously have an available copy on a provider to ensure fast execution. A high parallelism factor models that an application runs parallel tasks on the same data, which also requires multiple replicas. These two factors, ranging from 0 to 1, may be transmitted to the data management system by the application itself or observed by the meta-data manager at runtime. The equations for calculating the three coefficients C_{data}, C_{cap}, and C_{app} are as follows:

$$C_{data} = \frac{s_d}{s_{max}}; C_{cap} = \frac{\sum_{i=1}^{k} c_{f_i}}{\sum_{i=1}^{k} c_{t_i}}; C_{app} = \frac{f_{ava} + f_{par}}{2} \quad (1)$$

The relative importance of the four context coefficients can vary, depending on system and application state. Therefore, we multiply weights α_1 to α_4 to the coefficients. The sum of these weights further determines the maximum number of replicas in the system. In edge environments, k, the number of devices in the system, changes and does not necessarily equal the norm size of the system that was used during design time (k_0). Hence, we scale the resulting number of replicas depending on the relative size of the current system compared to the norm size. The final equation to calculate the number of replicas n then looks as follows:

$$n = (\alpha_1 * C_{data} + \alpha_2 * C_{cap} + \alpha_3 * C_{flu} + \alpha_4 * C_{app}) * \frac{k}{k_0} \quad (2)$$

The weights allow programmers to adjust the replication strategy to the needs of their system. For instance, in systems with large bandwidths that focus on fast execution of time-critical tasks, α_1 might be positive. With increasing data size, the number of replicas also increases as migrating aborted tasks becomes particularly costly. Systems with low bandwidth or limited storage capacities may use a negative α_1 to decrease the data transfer overhead. A proper choice of the weights is crucial for the effectiveness of the replication strategy. Unsuitable weights may lead to unsatisfactory results such as a constant shortage of replicas. Using a control structure that adapts the weights dynamically at runtime can reduce the risk of choosing unsuitable weights in the first place.

So far, we have considered context dimensions of the data and the system itself to determine an adequate number of replicas. For the decision on where to store the replicas, we also take the characteristics of the providers into account. If possible, replicas should be stored on devices that will not leave the system until the data transfer was worthwhile. Thus, the *stability* of the providers is an important context variable. We already identified the influence of stability in another area of application in our fault-avoidance approach [41]. Similar to our prior work, we characterize the stability by considering mean μ_p and variance σ_p^2 of the devices' residence times. The *mean residence time* is a relevant context dimension as it helps to predict whether a resource remains connected for a longer time on average. However, some devices may be connected for a long time on average but still leave the system after a short while in some cases. To consider this behavior in the decision, we add the *variance of the residence time* as a context variable to distinguish stable resources from unpredictable resources. The meta-data manager estimates the values for μ_p and σ_p^2 based on the past residence times of the resources in the system. Further, it monitors the current *residence time* t_p.

In addition, the *storage load* c_p of the providers is a relevant context variable. If a device faces a high storage load, replicating additional data on this resource may not be beneficial. Further, the scheduler also takes the *data queue sizes* q_p of the providers into account. Devices that store a large amount of replicas will likely run a larger number of tasks in the future. New replicas should be stored on devices that store less data to avoid task queues and to allow a timely execution of the associated task for this replica. Finally, the *relative performance index* RPI of a device determines its computational performance compared to the average performance of the current environment based on a benchmark measurement. A provider with a high RPI stores more data replicas due to the processing performance. Combining all of these context variables into an equation to calculate a provider's utility U_p leads to the following function:

$$U_p = \beta_1 * (\mu_p - t_p) + \beta_2 * \sigma_p^2 + \beta_3 * c_p - \beta_4 * q_p + \beta_5 * RPI \quad (3)$$

Context-aware replication now places the new replicas on the n providers with the highest utility value. Similar to Equation 2, the different context variables are attached with weights β_1 to β_5 to allow a customization of the function. Since the choice of relevant context dimensions does not claim to be exhaustive for all use cases, Equations 2 and 3 both allow to integrate further context dimensions as addends if required.

B. Task Scheduling Level

After the data placement level has distributed data in the system, the task scheduling level decides on the devices for the task execution. Here, we present the scheduling strategies *random task scheduling*, *data-aware scheduling*, and *performance-aware scheduling*. All strategies allow to parallelize tasks on multiple devices if desired. If no replica is present on a provider or data and tasks entered the system

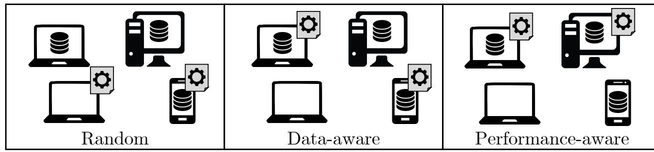


Fig. 5. The three algorithms that are integrated in the task scheduling level of the multi-level scheduler. The first strategy distributes task randomly, the second strategy considers the current data placement, and the third algorithm additionally considers the providers' performance.

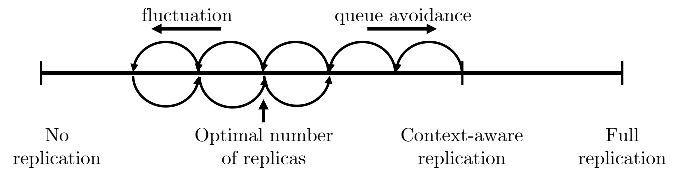


Fig. 6. The dynamic runtime adaptation approximates the optimal number of data replicas step by step. In case of device fluctuation, the number of replicas is decreased. If queues emerge, new replicas are placed in the system.

simultaneously, data is automatically coupled to the task. In case of a device failure or a task abortion, the scheduler re-allocates the whole task by applying the chosen strategy.

1) Random Task Scheduling: The random task scheduling chooses resource providers without applying any contextual knowledge. This algorithm does not take the data placement into account. Thus, tasks are scheduled randomly which may imply an additional data transfer, even if the data is present on other providers in the system.

2) Data-aware Task Scheduling: The data-aware scheduling strategy randomly picks a provider that has a local replica of the required data. This may lead to task queues if only few providers store a replica. If no provider stores a replica, the scheduler allocates the task to a random provider and transfers the required data together with the task.

3) Performance-aware Task Scheduling: With performance-aware task scheduling, the scheduler allocates tasks on the fastest idle device that holds a replica. To achieve this, we compare the aforementioned relative performance indices of all devices to determine the provider with the best performance. In case that all devices are busy, the scheduler randomly allocates the task on a provider with the required replica, similar to data-aware scheduling. Again, this may lead to task queues on resource providers.

C. Runtime Adaptation Level

The runtime adaptation level observes the system state during the runtime of tasks. So far, the number of replicas is not changed after the initial placement. Fluctuation implies a decreasing amount of data replicas over time, which may introduce execution latencies. As a solution, the scheduler adjusts the data placement during runtime. Here, we implemented the strategies *static runtime adaptation*, *dynamic runtime adaptation*, and *data caching*.

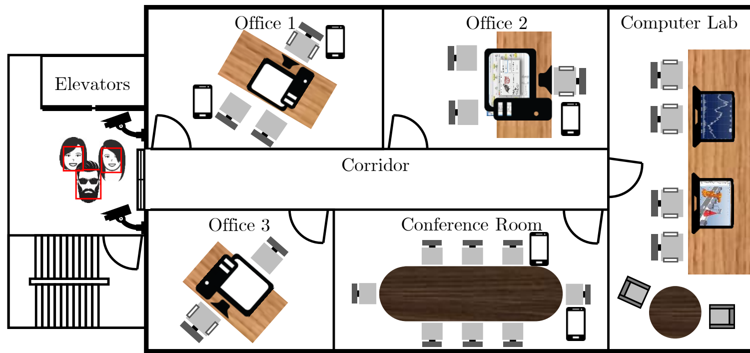
1) Static: A first approach is to restore the amount of replicas chosen by the initial data placement strategy. In case of the 1-replication strategy, the runtime data placement always keeps a single replica. When the provider that holds the replica leaves the system, the runtime adaptation level allocates a new replica. Analogously, this strategy restores the full and context-aware replication in case of fluctuation.

2) Dynamic: The goal of the context-aware replication strategy is to determine an appropriate number of replicas and to store these replicas on most suitable providers. The decision depends on multiple, constantly changing context variables. To take context changes into account, the dynamic runtime adaptation identifies the current context and adjusts the amount of replicas n. However, determining a suitable value for n is both, crucial and complex. On the one hand, if n is exaggerated, the overhead of managing the replicas is increased. On the other hand, if n is understated, the responsiveness of task executions decreases.

To approximate the number without a large amount of optimization overhead, we apply a MAPE feedback loop from self-adaptive systems to our approach [42]. The MAPE feedback loop has four components, namely, *monitoring*, *analyzing*, *planning*, and *executing* that realize different tasks in the system: To measure the execution quality, we *monitor* the queuing times of tasks relative to their execution time. The system *analyzes* the queuing time measurements and verifies if a threshold is exceeded. This triggers the creation of a new replica. Now, the *planning* stage of the feedback loop uses the utility function introduced in Equation 3 to determine the most suitable resource provider. In the *execution* phase, the data management system deploys a new replica on the chosen resource provider. This feedback loop leads to a work balancing among all providers with the same data.

With this algorithm, we increase the number of replicas if required. However, too many replicas lead to data management overhead. As a solution, the natural fluctuation of edge devices decreases the number of replicas in the system. Thus, we approximate the optimal n by applying our dynamic runtime adaptation and relying on the natural fluctuation of edge devices. Depending on the current data placement strategy, the approximation to the optimal n may start with a 1-replication, a full replication, or a context-aware replication.

3) Data Caching: With the caching strategy enabled, resource providers always store incoming data for further use as local replica. When no initial replication strategy is active and tasks and data are allocated together, this strategy reduces the data transfer overhead. Each resource provider decides if it has enough storage for an additional replica. In case of random task scheduling, the data placement converges towards a full replication over time. The strength of this effect depends on the stability of the environment. Besides, the emerging data placement is rather random and not based on any strategy, which requires a sophisticated garbage collection mechanism. Therefore, caching may introduce additional data management overhead, depending on the current system context.



Device	Bench-Mark	μ (in h)	σ (in h)
3 x PC	7	7	2.5
2 x Laptop	10	4	2.7
3 x Phone (stable)	20	1.5	1
2 x Phone (unstable)	20	0.5	0.5

Application	# Tasks	Runtimes (in sec)	Data (in MB)
Face Detection	243	4	120
Machine Learning	40	30, 60, 120,180	10, 15, 20, 25, 30
MC Simulation	20	300	5
Combined	303	-all-	-all-

Fig. 7. The testbed used in the evaluation represents a typical office environment in the academic sector. Ten devices act as resource providers with different characteristics concerning benchmark, mean residence time μ and standard deviation of the residence time σ. In the evaluation, this distributed computing environment runs three workflows typical for applications in this area and a combined workflow with all applications in parallel.

V. Evaluation

We evaluate our multi-level scheduling approach in a real-world testbed. The usage of a real-world testbed allows to show the practical feasibility of the approach. Moreover, environmental details and realistic characteristics of the network layer influence the measurements. This comes at the cost of less controllability and natural variances compared to a simulator-based evaluation. After presenting our data management prototype, the experimental setup, and the evaluation scenarios, this section summarizes the results of an evaluation of the three levels of our scheduling approach.

A. Prototype

We implemented a prototype of our data management system in Java. The implementation offers well-defined interfaces that allow to integrate the data management system into different distributed computing systems. In the evaluation, we use our data management approach with the *Tasklet* system, a middleware-based distributed computing system for heterogeneous environments [43]. To avoid possible influences of performance fluctuations or side effects of the external middleware, we connected our prototype to a Tasklet system emulator. This emulator shows equivalent behavior as the Tasklet system but allows a more controlled and steady setup.

Tasklets are extracted application subroutines that are offloaded to remote resources in self-contained units. The Tasklet system model includes three entities. In addition to resource consumers and providers, certain nodes in the network act as *brokers*. After registering at one broker, the consumers submit task requests. The broker then schedules tasks and orchestrates the workflow. The Tasklet system allows nodes to be consumers and providers at the same time. Figure 8 shows the system model of the Tasklet system [43]. To tailor the data management system to the hybrid architecture of the Tasklet system, we deployed the multi-level scheduler, the meta-data manager, and the garbage collector centrally on the broker.

B. Experimental Setup

To deploy the prototype, we created a real-world testbed consisting of eleven physical devices. One of the devices acts as the consumer and hosts the broker. The other ten devices are resource providers. To exclude hardware influences, we used homogeneous devices and configured them to have the characteristics of desktop PCs, laptops, and smartphones in terms of computational performance and fluctuation. Now, the setup resembles an office environment in the academic sector with three office rooms and a student lab depicted in Figure 7. Leaving devices delete all of their data and re-enter the system after a randomly chosen time interval. In this setup, we run three applications separately for 60 minutes for each possible combination of data placement strategy, task scheduling strategy, and runtime adaptation mechanism. Additionally, we combine them in a fourth 60-minutes-workflow to model a simultaneous execution of all three applications. The workflows contain between 20 and 303 Tasklet executions each. In total, the evaluation time amounted to 48 hours.

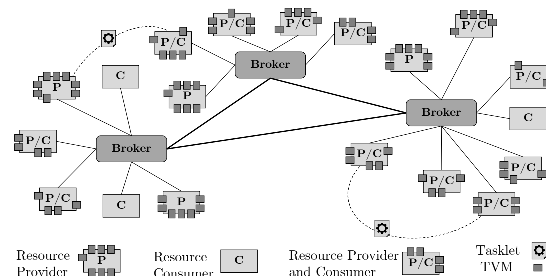


Fig. 8. The Tasklet system model. In addition to resource consumers and providers, central brokers exist that schedule remote computations. Providers run Tasklet Virtual Machines (TVMs) to execute Tasklets.

To compare the performance of the strategies, we measure the Tasklet turnaround time, the queuing time, the execution time, and the data transfer overhead. The queuing time quantifies the time span between the arrival of the Tasklet at the provider and the execution start.

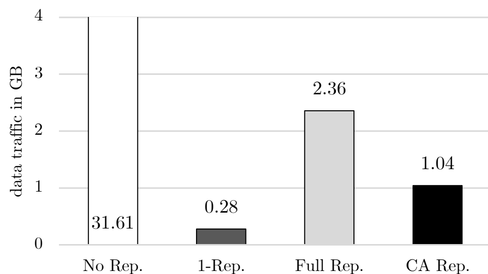


Fig. 9. The initial data transfer overhead for creating a 1-replication, a full replication, and a context-aware replication in comparison to the overhead of transferring the data coupled with the tasks.

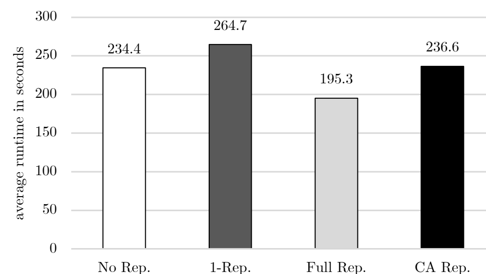


Fig. 10. The average turnaround times for each of the four data placement strategies if random task scheduling is applied. Runtime adaptation is disabled.

C. Scenarios

We run the prototype with three different applications of varying data and computational intensity in the academic office environment. First, a *face recognition* workflow consists of events that occur if cameras at the entrance of the office building use face recognition to grant access. Face recognition compares the current camera image to entries of a comparatively large database file. Second, a researcher in office 2 tests different *machine learning* algorithms on multiple input files of varying sizes. Third, students in the computer lab perform computationally intensive *simulations* on comparatively small input files. A poisson point process is used to generate realistic timing of task events in these workflows. To enhance comparability of data-centric scheduling strategies and to avoid unintended variations in the execution of the Tasklets, the prototype uses *emulated Tasklets* with fixed complexities.

D. Data Placement Level

The data placement level optimizes the data distribution before task runtime. This initial placement leads to an additional data transfer overhead. Figure 9 shows the total amount of data shipped from the consumer to the providers for each data placement strategy in the scenario when all three applications run in parallel. It becomes visible that the full replication strategy, which is most promising from a runtime perspective, also leads to high data transfer overhead in the pre-execution phase. In case of data-aware scheduling and deactivated runtime adaptation, no data transfers occur during runtime for 1-replication, full replication, and context-aware replication. Contrary, the no replication strategy does not require initial data transfers but leads to a considerable amount of transferred data during runtime. Since the number of tasks is higher than the number of providers, coupling data and tasks in the no replication strategy is considerably more costly in terms of data transfer compared to any other strategy.

E. Task Scheduling Level

After having distributed the data in the system according to the data placement strategy, the task scheduling level allocates task to resources. First, we apply the random task scheduling strategy as a baseline. Figure 10 depicts the average Tasklet turnaround times for each of the four data placement strategies. As expected, a full replication strategy performs best in this setting. Since all providers already store the data, no data transfers are necessary and turnaround times only consist of queuing time and execution time. No replication, 1-replication, and context-aware replication perform substantially worse.

Now, we use data-aware task scheduling instead of random scheduling. This strategy exploits the distribution of replicas created by the data placement level. Figure 11 shows the average turnaround times for the combined workflow. Data-aware scheduling is only applicable to 1-replication and context-aware replication. Compared to random task scheduling, a data-aware strategy reduces the turnaround times by 56.4 % for context-aware replication. The considerable improvement indicates that it is highly beneficial to apply a data-centric task scheduling strategy if and only if the data placement level already optimized data placement in advance. However, when using the 1-replication strategy, a data-aware task scheduling strategy substantially extends turnaround times. All tasks are scheduled on the single device which leads to tasks queues.

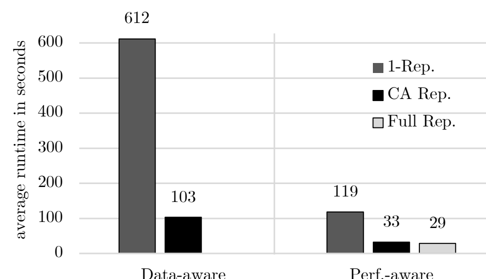


Fig. 11. Tasklet turnaround times for the different data placement strategies combined with data-aware and performance-aware task scheduling.

As a third option, we apply performance-aware task scheduling. When using context-aware replication, this further decreases the Tasklet turnaround times by 68.3 % compared to data-aware task scheduling with the same placement strategy. As depicted in Figure 11, the combination of context-aware data replication and performance-aware task scheduling now reaches a similar system performance as the runtime-optimal combination of a constant full replication and performance-aware task scheduling. Average turnaround times are 32.7

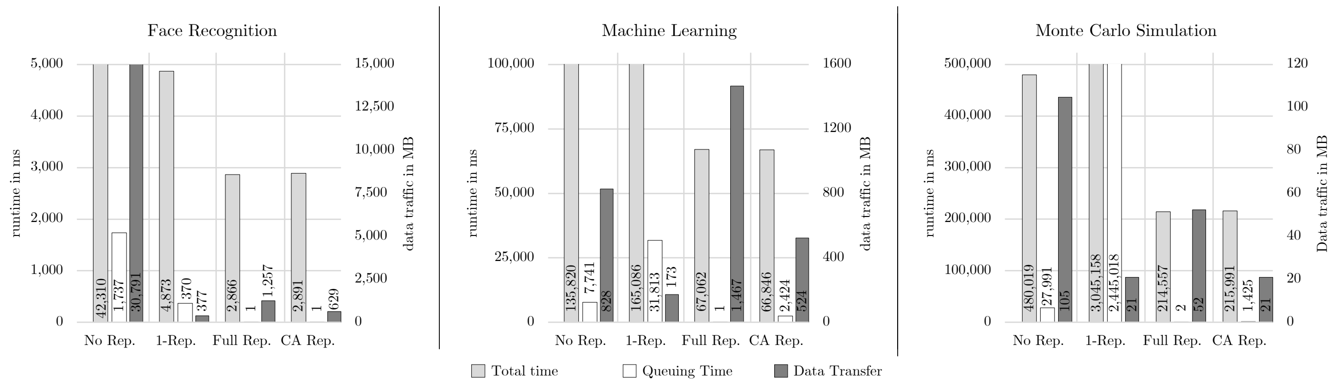


Fig. 12. Turnaround times, queuing times, and data transfer overhead for the different data placement strategies if each of the three applications runs separately in the environment. Placement strategies are combined with performance-aware task scheduling and dynamic runtime adaptation when applicable.

seconds compared to 28.8 seconds in a full replication. Thus, the context-aware replication together with appropriate task allocation can be comparably as effective as a full replication while requiring a substantially lower data transfer overhead even without any runtime adaptation.

F. Runtime Adaptation Level

To further improve the performance, we now also activate runtime adaptation. When used with context-aware replication and performance-aware task scheduling, dynamic runtime adaptation minimizes turnaround times by another 10.2 % to 29.4 seconds. Hence, the top-performing combination of context-aware replication, performance-aware task scheduling, and dynamic runtime adaptation leads to runtimes comparable to a constant full replication while reducing data overhead.

G. Applications Spotlight

The effectiveness of data placement and task scheduling strategies is highly dependent on the nature of the current workflow. To investigate the effect of application characteristics, we evaluate our data placement strategies for each application separately. Figure 12 presents the average turnaround times, queuing times, and data transfer overhead. If possible, our data placement strategies are combined with performance-aware task scheduling and dynamic runtime adaptation. The face recognition application is highly data-intensive while requiring only short computations. Here, task queues only emerge in the no replication strategy due to the high data transfer overhead during runtime. However, the 1-replication strategy is still considerably slower than context-aware and full replication as a slower device stores the only replica.

The machine learning use case relies on smaller input files than the face database in the first application but is more computationally intensive. If this application runs isolated, large task queues emerge on the devices that store the only replica in the 1-replication strategy. In this case, even the no replication strategy performs better. Similar to the face recognition use case, context-aware replication is again able to meet the performance of a full replication while requiring only 35.7 % of the data transfer overhead.

The third use case is the execution of a simulation. This application runs complex computations with small input data. Again, context-aware replication performs as well as full replication. Together with performance-aware task scheduling and dynamic runtime adaptation, the strategy is able to keep queuing times at a low level. Thus, it is as fast as a full replication while leading to considerably less data transfer overhead. The 1-replication strategy, however, fails to eliminate task queues even though it uses dynamic runtime adaptation. The reason for this behavior is that the initial distribution of the data is too narrow for performance-aware task scheduling and that the dynamic adaptation is not able to cope with the long queues which evolve after a short time. Here, a solution would be to re-configure the dynamic runtime adaptation to create more replicas at once and to be less conservative in scenarios with low data intensity.

VI. Conclusion

In this paper, we introduced a data management approach for distributed computing in edge environments. We proposed a multi-level scheduler that places data strategically in the system based on the current context, schedules tasks accordingly, and adapts data placement at runtime. A prototype of the system tailored to the Tasklet system was evaluated in a real-world testbed. We investigated the effectiveness of our proposed algorithms. The results show that a context-aware replication strategy together with performance-aware task scheduling and dynamic runtime adaptation performs best. It leads to task turnaround times comparable to a full replication while requiring less data overhead.

For future work, we plan to integrate the data management architecture into the Tasklet simulator to evaluate the approach on a larger scale. Further, we want to extend the data placement level of our scheduler and incorporate more context dimensions such as network topology or user preferences.

Acknowledgment

This work was supported by the German Research Foundation (DFG) and the Julius Paul Stiegler Memorial Foundation.

REFERENCES

[1] R. Hasan, M. M. Hossain, and R. Khan, "Aura: An IoT Based Cloud Infrastructure for Localized Mobile Computation Outsourcing," in *Proc. MobileCloud*. IEEE, 2015, pp. 183–188.

[2] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama, and R. Buyya, "A Context Sensitive Offloading Scheme for Mobile Cloud Computing Service," in *Proc. CLOUD*. IEEE, 2015, pp. 869–876.

[3] A. Mtibaa, K. A. Harras, and A. Fahim, "Towards Computational Offloading in Mobile Device Clouds," in *Proc. CloudCom*. IEEE, 2013, pp. 331–338.

[4] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: "Better than Best-Effort" Computing," in *Proc. ICCCN*. IEEE, 2016, pp. 1–11.

[5] A. H. Alhusaini, V. K. Prasanna, and C. Raghavendra, "A Unified Resource Scheduling Framework for Heterogeneous Computing Environments," in *Proc. HCW*. IEEE, 1999, pp. 156–165.

[6] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling Data-Intensive Workflows onto Storage-Constrained Distributed Resources," in *Proc. CCGrid*. IEEE, 2007, pp. 401–409.

[7] T. Kokilavani and D. George Amalarethinam, "Load Balanced Min-Min Algorithm for Static Meta-Task Scheduling in Grid Computing," *Int. J. Comput. Appl.*, vol. 20, no. 2, pp. 42–48, 2011.

[8] D. Yuan, Y. Yang, X. Liu, and J. Chen, "A Data Placement Strategy in Scientific Cloud Workflows," *Futur. Gener. Comput. Syst.*, vol. 26, no. 8, pp. 1200–1214, 2010.

[9] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing," in *Proc. CCGRID*. IEEE, 2011, pp. 295–304.

[10] R. Van Den Bossche, K. Vanmechelen, and J. Broeckhove, "Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds," *Futur. Gener. Comput. Syst.*, vol. 29, no. 4, pp. 973–985, 2013.

[11] O. Choudhury, D. Rajan, N. Hazekamp, S. Gesing, D. Thain, and S. Emrich, "Balancing Thread-level and Task-level Parallelism for Data-Intensive Workloads on Clusters and Clouds," in *Proc. ICCC*. IEEE, 2015, pp. 390–393.

[12] D. Schäfer, J. Edinger, M. Breitbach, and C. Becker, "Workload Partitioning and Task Migration to Reduce Response Times in Heterogeneous Computing Environments," in *Proc. ICCCN*. IEEE, 2018, (to be published).

[13] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A Comparison of Eleven Static Mapping Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *J. Parallel Distrib. Comput.*, vol. 61, pp. 810–837, 2001.

[14] X. He, X. Sun, and G. von Laszweski, "QoS Guided Min-Min Heuristic for Grid Task Scheduling," *J. Comput. Sci. Technol.*, vol. 18, no. 4, pp. 442–451, 2003.

[15] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid," in *Proc. SC*. ACM/IEEE, 2000, pp. 111–126.

[16] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task Scheduling Strategies for Workflow-based Applications in Grids," in *Proc. CCGRID*. IEEE, 2005, pp. 759–767.

[17] R. McClatchey, A. Anjum, H. Stockinger, A. Ali, I. Willers, and M. Thomas, "Scheduling in Data Intensive and Network Aware (DIANA) Grid Environments," *J. Grid Comput.*, vol. 5, no. 1, pp. 43–64, 2007.

[18] H. Liu, A. Abraham, V. Snášel, and S. McLoone, "Swarm scheduling approaches for work-flow applications with security constraints in distributed data-intensive computing environments," *Inf. Sci. (Ny).*, vol. 192, pp. 228–243, 2012.

[19] J. Taheri, Y. Choon Lee, A. Y. Zomaya, and H. J. Siegel, "A Bee Colony based optimization approach for simultaneous job scheduling and data replication in grid environments," *Comput. Oper. Res.*, vol. 40, no. 6, pp. 1564–1578, 2013.

[20] I. Casas, J. Taheri, R. Ranjan, L. Wang, and A. Y. Zomaya, "A balanced scheduler with data reuse and replication for scientific workflows in cloud computing systems," *Futur. Gener. Comput. Syst.*, vol. 74, pp. 168–178, 2017.

[21] X. Li, T. Jiang, and R. Ruiz, "Heuristics for periodical batch job scheduling in a MapReduce computing framework," *Inf. Sci. (Ny).*, vol. 326, pp. 119–133, 2016.

[22] M. S. Elbamby, M. Bennis, and W. Saad, "Proactive Edge Computing in Latency-Constrained Fog Networks," in *Proc. EuCNC*. IEEE, 2017, pp. 1–6.

[23] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data Management and Transfer in High-Performance Computational Grid Environments," *Parallel Comput.*, vol. 28, no. 5, pp. 749–771, 2002.

[24] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini, "Simulation of Dynamic Grid Replication Strategies in OptorSim," *Int. J. High Perform. Comput. Appl.*, vol. 17, no. 4, pp. 403–416, 2002.

[25] S. Venugopal, R. Buyya, and L. Winton, "A Grid service broker for scheduling e-Science applications on global data," *Concurr. Comput. Pract. Exp.*, vol. 18, pp. 685–699, 2006.

[26] A. Chervenak, E. Deelman, M. Livny, M.-H. Su, R. Schuler, S. Bharathi, G. Mehta, and K. Vahi, "Data Placement for Scientific Applications in Distributed Environments," in *Proc. GRID*. IEEE, 2007, pp. 267–274.

[27] D. Thain, T. Tannenbaum, and L. Miron, "Distributed Computing in Practive: The Condor Experience," *Concurr. Comput. Pract. Exp.*, vol. 17, no. 2, pp. 325–356, 2005.

[28] T. Kosar and M. Balman, "A new paradigm: Data-aware scheduling in grid computing," *Futur. Gener. Comput. Syst.*, vol. 25, no. 4, pp. 406–413, 2009.

[29] D. T. Nukarapu, B. Tang, L. Wang, and S. Lu, "Data Replication in Data Intensive Scientific Applications with Performance Guarantee," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 8, pp. 1299–1306, 2011.

[30] K. Ranganathan and I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications," in *Proc. HPDC*. IEEE, 2002, pp. 352–358.

[31] F. Desprez and A. Vernois, "Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid," *J. Grid Comput.*, vol. 4, no. 1, pp. 19–33, 2006.

[32] M. Tang, B. S. Lee, X. Tang, and C. K. Yeo, "The impact of data replication on job scheduling performance in the Data Grid," *Futur. Gener. Comput. Syst.*, vol. 22, pp. 254–268, 2006.

[33] A. Chakrabarti and S. Sengupta, "Scalable and Distributed Mechanisms for Integrated Scheduling and Replication in Data Grids," in *Proc. ICDCN*. Springer, 2008, pp. 227–238.

[34] R.-S. Chang, J.-S. Chang, and S.-Y. Lin, "Job scheduling and data replication on data grids," *Futur. Gener. Comput. Syst.*, vol. 23, pp. 846–860, 2007.

[35] B. Tang, Z. Chen, G. Hefferman, S. Pei, T. Wei, H. He, and Q. Yang, "Incorporating Intelligence in Fog Computing for Big Data Analysis in Smart Cities," *IEEE Trans. Ind. Informatics*, vol. 13, no. 5, pp. 2140–2150, 2017.

[36] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed Edge Cloud for Data Intensive Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3229–3242, 2017.

[37] Y. Li, J. Luo, J. Jin, R. Xiong, and F. Dong, "An Effective Model for Edge-Side Collaborative Storage in Data-Intensive Edge Computing," in *Proc. CSCWD*. IEEE, 2018, pp. 535–540.

[38] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling Remote Computing Among Intermittently Connected Mobile Devices," in *Proc. MobiHoc*. ACM, 2012, pp. 145–154.

[39] M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data-intensive computing," in *Proc. CTS*. IEEE, 2014, pp. 491–492.

[40] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 4th ed. Addison-Wesley, 2005.

[41] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker, "Fault-avoidance strategies for context-aware schedulers in pervasive computing systems," in *Proc. PerCom*. IEEE, 2017, pp. 79–88.

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

[43] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: Overcoming Heterogeneity in Distributed Computing Systems," in *Proc. ICDCSW*. IEEE, 2016, pp. 156–161.