UNIVERSITÄT
MANNHEIM

Elastic Computation Placement in Edge-based Environments

Inauguraldissertation

zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
der Universität Mannheim

vorgelegt von

Bernd Dominik Schäfer

aus Kassel

Dekan: Prof. Dr. Dieter Truxius
Referent: Prof. Dr. Christian Becker
Korreferent: Prof. Dr. Gregor Schiele

Tag der mündlichen Prüfung: 07. Februar 2019

Abstract

Today, technologies such as machine learning, virtual reality, and the Internet of Things are integrated in end-user applications more frequently. These technologies demand high computational capabilities. Especially mobile devices have limited resources in terms of execution performance and battery life. The offloading paradigm provides a solution to this problem and transfers computationally intensive parts of applications to more powerful resources, such as servers or cloud infrastructure. Recently, a new computation paradigm arose which exploits the huge amount of end-user devices in the modern computing landscape – called edge computing. These devices encompass smartphones, tablets, microcontrollers, and PCs. In edge computing, devices cooperate with each other while avoiding cloud infrastructure. Due to the proximity among the participating devices, the communication latencies for offloading are reduced. However, edge computing brings new challenges in form of device fluctuation, unreliability, and heterogeneity, which negatively affect the resource elasticity.

As a solution, this thesis proposes a computation placement framework that provides an abstraction for computation and resource elasticity in edge-based environments. The design is middleware-based, encompasses heterogeneous platforms, and supports easy integration of existing applications. It is composed of two parts: the Tasklet system and the edge support layer. The Tasklet system is a flexible framework for computation placement on heterogeneous resources. It introduces closed units of computation that can be tailored to generic applications. The edge support layer handles the characteristics of edge resources. It copes with fluctuation and unreliability by applying reactive and proactive task migration. Furthermore, the performance heterogeneity and the consequent bottlenecks are handled by two edge-specific task partitioning approaches. As a proof of concept, the thesis presents a fully-fledged prototype of the design, which is evaluated comprehensively in a real-world testbed. The evaluation shows that the design is able to substantially improve the resource elasticity in edge-based environments.

Acknowledgments

First of all, I would like to thank my supervisor Prof. Dr. Christian Becker for the continuous support since I became a Bachelor's degree candidate in 2010. Christian, thank you for your guidance and the opportunity to start my academic career in your group. Thanks for the motivation, the encouragement, and the freedom to follow my own path, but also for the hard questions that helped me to keep on track. Furthermore, thank you for the fun times we had on our conference trips all around the world, the food and wine education, and for dragging me through the Chinese security. I will miss being a member of "the cruiser".

I would like to thank Prof. Dr. Gregor Schiele for his insightful comments and the willingness to act as the second reviewer. Gregor, thank you for trusting me as a student assistant in the first place, teaching me the path to clean code, and for always having a good advise at hand.

I would like to thank Prof. Dr. Hartmut Höhle for joining the board of examiners as well as for bringing the Bourgogne to L15.

I would like to thank Dr. Justin Mazzola Paluska, for his valuable input and the opportunity to present my work at the MIT.

I would like to thank all the people I worked with at the chair, namely Dr. Patricia Arias-Cabarcos, Martin Breitbach, Janick Edinger, Kerstin Goldner, Benedikt Kirpes, Sonja Klingert, Dr. Christian Krupitzer, Markus Latz, Jens Naber, Martin Pfannemüller, Dr. Vaskar Raychoudhury, Dr. Felix Maximilian Roth, Dr. Sebastian VanSyckel, and Anton Wachner. You are the best team imaginable with the greatest spirit and camaraderie. It is a great pleasure and motivation to work with friends instead of colleagues. Especially, I would like to thank Janick for being my wingman during all those years. The countless hours of coding, evaluating in the lab, brainstorming at night, and writing papers would have been inconceivable without you. Thank you Sebastian – a.k.a. the red pencil – for teaching me how and how not to write papers. Thanks to Martin

(Breiti) for bringing fresh spirit into the Tasklet project, for following the Tasklet paper writing traditions, and for proofreading this thesis. Thank you Max, for the recreational guitar jam breaks and for offering paper writing support at night. Thanks to Christian (Pitzi) for sharing all your insights on the bureaucratic part of earning a PhD and for proofreading parts of this thesis. Thank you Jens, for being a fellow coffee addict. You all had an open ear for me during the final phase of my thesis.

This work was supported by the German Research Foundation (DFG) under grant BE 2498/10-1 'Tasklets: Ein Ansatz für Best-Effort Computing'. I would like to thank each thesis student involved in the project for their contributions, namely Tobias Borlinghaus, Martin Breitbach, Jens Eckrich, Tim Endlich, and Johannes Lang.

Last but not least, I would like to thank my family and friends for always being there for me. To my parents Anne and Manfred, thank you for your support, for raising me to be the person I am today, and for giving me the opportunity to go my own way. To my brother Oliver, thank you for encouraging me and being the best big brother to look up to. To my sister-in-law Susanne, my niece Leonie, my nephew Max, and my grandma Lina, thank you for keeping your fingers crossed and for always motivating me. To my wife Regina, thank you for your unconditional love, your support, and your patience. You gave me the strength I needed to pursue my work and went with me through ups and downs.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BOINC	Berkeley Open Infrastructure for Network Computing
BoT	Bag of Task
EC2	Elastic Compute Cloud
ETSI	European Telecommunications Standards Institute
FPGA	Field Programmable Gate Array
GO	Group Owner
GPU	Graphics Processing Unit
GTVM	GPU-based Tasklet Virtual Machine
GUI	Graphical User Interface
IoT	Internet of Things
IP	Internet Protocol
JNI	Java Native Interface
MBS	Mandelbrot Set
MDC	Mobile Device Cloud
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
MT	Microtasking
PAM	Performance-aware Microtasking
PAP	Performance-aware Partitioning
QoC	Quality of Computation
QoS	Quality of Service
RPC	Remote Procedure Call
RTT	Round Trip Time
SoC	System on a Chip
TCP	Transmission Control Protocol
TVM	Tasklet Virtual Machine
TVMM	Tasklet Virtual Machine Manager
UDP	User Datagram Protocol

1. Introduction

The modern computing landscape consists of a vast amount of heterogeneous end-user devices. From that, a new paradigm emerged, called edge computing. It describes the decentralized collaboration of end-user devices while avoiding the use of centralized infrastructure, such as the cloud. Recent edge devices are equipped with fairly powerful hardware that may runs idle. While this computing power remains unused, other devices work to full capacity and would benefit from additional computing resources. As a solution, computation placement executes computationally intensive parts of applications on remote resources and augments the computational capabilities of devices that are limited in performance.

The goal of the thesis is to design an elastic computation placement framework that is focused on edge resources. Thus, computation can seamlessly be exchanged among heterogeneous edge devices. Software developers write programs in any programming language without being aware of whether the computation is executed locally or on heterogeneous remote devices. Every device can share its computing power and contribute to a worldwide network of edge resources. As a result, the borders of physical devices vanish and computation becomes a common good that can be exchanged.

The remainder of this chapter motivates the thesis and states the problem as well as the research questions. Further, it gives an overview of the scientific contributions and the structure.

1.1. Motivation

Recent technologies such as machine learning, virtual reality, and the Internet of Things (IoT) require high computational performance and responsiveness. These technologies are embedded in common software running on mobile and desktop end-user devices. The resource demand of these applications, however, may exceed the performance that is locally available. As a solution, computation placement augments the computational capability of weak end-user devices to run applications with high performance requirements. It bundles up computationally intensive application parts and transfers them to a more powerful remote resource where the computation takes place. After that, the result is sent back to the application. This entire process is highly complex in terms of network handling, computation abstraction, or resource heterogeneity. As an abstraction, middleware developers create software that hides these complexities and provides an easy-to-use application programming interface (API) to place computation.

1.2. Problem Statement

Traditional offloading systems place the computation on rather static infrastructures like dedicated servers [48], computational grids [5], or clouds [55]. Cloud resources are elastic, meaning, that the provisioning of resources adapts to the current performance demand [93]. While offering nearly unlimited resources, the cloud has major drawbacks in terms of trust, security, privacy [102, 207], and communication latency [24].

As an alternative, user-controlled edge devices can serve as computational resources. The modern computing landscape consists of a plethora of heterogeneous computing entities at the edge, all serving a certain purpose. It includes devices such as smartphones, tablets, laptops, and PCs, equipped with graphics processing units (GPUs). In case these devices run idle, they can contribute their computational performance to a global distributed computing system. Compared to cloud resources, the edge exploits the locality of resources leading to short communication latencies and a high responsiveness for task executions. Moreover, edge devices potentially provide more trust, security, and privacy, if they are not hosted by a third party. Apart from that, devices in the edge have several drawbacks such as limitations in performance and reliability, fluctuation and errors, as well as different kinds of heterogeneities. To cope with errors and fluctuation, edge-specific fault tolerance mechanisms are required to increase the reliability of end-user devices. Further, a computation abstraction in combination with a middleware-based system solution is needed to overcome heterogeneities. The performance of edge environments is, however, limited by nature since it depends on the number of participating devices in proximity. The integration of specific computing architectures, like GPUs, and an efficient scheduling approach are required to exploit the maximum performance of the available resources.

1.3. Objective and Research Questions

Based on the motivation and the problem statement, the objective of the thesis is defined as: *The development of a computation placement framework that provides an abstraction for computation and resource elasticity in edge-based environments.* Derived from the objective, this thesis will answer the following research questions:

1. How can *elastic computation placement* in edge environments be achieved?
2. How is the *application model* of a computation placement system defined?
3. How can a *lightweight computation abstraction* be realized?
4. How do heterogeneous edge devices *perform in comparison*?
5. How can a system *compensate unreliability and heterogeneity* in the edge?

1.4. Thesis Contributions

To answer the research questions, this thesis presents an approach for elastic computation placement in edge computing environments. It includes a computation placement framework that abstracts computation to handle various types of heterogeneity. A middleware encompasses the assembly, distribution, and execution of closed computation units as well as edge-centric mechanisms to increase fault tolerance and resource efficiency. Based on a literature review, the research gap is identified and the system design is developed. This design is implemented in a fully-fledged prototype which is the foundation for an evaluation in a real-world testbed. The six main contributions of the thesis are as follows:

(i) Analysis of the State of the Art: The thesis presents an exhaustive analysis of the state of the art in distributed computing systems. Related approaches from areas such as cluster, grid, volunteer, cloud, fog, and edge computing are investigated. For the literature analysis, a classification is developed that is used to categorize the existing approaches and to identify the research gap.

(ii) Application Model for Computation Placement: Not all applications benefit from computation placement. The second contribution is a model that classifies applications by means of different characteristics. Based on these characteristics, the model determines to what extent an application benefits from remote computation placement.

(iii) Lightweight Computation Abstraction: For the integration of heterogeneous computation platforms, the thesis defines a computation abstraction in form of closed units of computation – called Tasklets. Regardless of the computing architecture, the abstraction extracts the plain computational capabilities of a device. This includes PCs, smartphones, tablets, and GPUs. The abstraction is lightweight to incorporate thin devices, like microcontrollers and embedded devices.

(iv) Framework for Elastic Computation Placement on Edge Resources: The main contribution includes the design of a framework for elastic computation placement. It is middleware-based, encompasses heterogeneous platforms, and supports easy integration of existing applications. Further, it includes a performance measure for remote computation placement. This design especially considers the characteristics of edge devices, thus, it improves the fault tolerance and avoids performance bottlenecks. A so called edge support layer encompasses this functionality. It offers task migration and performance-aware workload partitioning algorithms that are specialized for edge resources.

(v) Comprehensive Prototype Implementation: The thesis contributes a fully-fledged prototype implementation of the presented design. The prototype integrates the creation, distribution, and execution of closed computation units as well as the edge support layer. For testing and evaluating purposes, several applications from different domains were implemented for the system. While these applications conform to the application model, they have different placement characteristics in terms of data and computation intensity.

(vi) Evaluation in a Real-World Testbed: Finally, the thesis includes an exhaustive evaluation of the prototype by means of several experiments. These experiments use different applications and environment settings. All settings consist of a real-world testbed with different combinations of devices to show specific behavior and characteristics of the system.

1.5. Structure

This thesis is structured as follows: After the introduction, Chapter 2 describes the theoretical background and the terminology of the thesis. This includes the areas distributed computing systems, computation placement, and pervasive computing. Then, Chapter 3 derives the functional and non-functional requirements of the thesis by means of a scenario. In Chapter 4, related work is classified and the research gap is identified. Afterwards, Chapter 5 presents the design of the two-layered research approach. It consists of the framework for computation placement – the Tasklet system – and the edge support layer. Based on that, Chapter 6 describes the implementation of the full-fledged prototype. After that, the approach is evaluated in Chapter 7 by means of seven experiments. These experiments include a qualitative evaluation of the requirements from Chapter 3 and a quantitative evaluation of all system parts. Finally, the thesis is closed in Chapter 8 with a conclusion and an outlook on future work.

2. Background

This section introduces the background and the fundamental concepts of relevant research areas of the thesis. After a brief definition of distributed systems, various types of distributed computing systems are presented. Then, the section introduces computation placement as a paradigm to execute computationally intensive tasks on remote machines. Lastely, context-aware computing is discussed to incorporate environmental information into task distribution strategies. In order to exploit user-controlled devices at the edge, the concept of context-awareness is crucial.

The term **distributed systems** has been defined in literature several times. Van Steen and Tanenbaum argue that none of these definitions are satisfactory and gave a loose characterization [184, p. 2]: *"A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system."* This definition emphasizes two main characteristics of distributed systems that are relevant for this thesis: First, it consists of autonomous collaborating elements that are connected by a network and second, for the user/application the system appears as a single entity. In addition to that, Coulouris *et al.* define distributed system in [47, p. 2] *"as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages."* Coulouris *et al.* further define challenges of these systems as heterogeneity, openness, security, scalability, failure handling, concurrency, and transparency [47, p. 16-25]. These challenges influence the design principles of the approach of this thesis. Distributed systems research is broad and includes a wide variety of types, such as the Internet [47, p. 3], mobile computing [47, p. 6], distributed information systems [184, p. 34], pervasive systems [184, p. 40], and distributed computing systems [184, p. 25]. The latter – also called high performance distributed computing more recently – is the main focus of the thesis and is therefore inspected in detail.

2.1. Distributed Computing Systems

A distributed computing system is a special distributed system focused on harvesting computational capabilities of remote resources. Several research areas emerged to reduce execution times of complex computing problems. Before the emerge of distributed computing, large and specialized supercomputers were the only option to obtain sufficient computing power. This solution, however, implies some drawbacks, like asset and maintenance costs as well as building complexity. As an alternative, a set of physical remote machines can operate as computing resource – the so called *resource providers* [32]. Assuming that a task can be split up into several independent parts, these parts can be executed on resource providers in the system. After the execution, the system collects and accumulates the result and forwards it to the application. This application is executed by the so called *resource consumer* [32]. This paradigm is used to speed up applications from various areas like scientific computing, economic predictions, machine learning, and video rendering.

In particular in particle physics, large amounts of data need to be processed. One example is the large hadron collider that records data of subatomic particles, which emerge from near lightspeed particle collisions. Each year, the large hadron collider generates roughly one peta byte of data [13], which is analyzed by a large compute cluster [121, 171]. This huge amount of data can only be processed by harvesting the computing power of a plethora of powerful devices. From this general idea, several research areas arose, which are *cluster*, *grid*, and *volunteer* computing. Over time, *cloud computing* evolved as a more centralized and service-oriented paradigm. Recently, with the appearance of powerful and mobile end-user devices, *mobile cloud*, *fog*, *mobile edge*, and *edge computing* emerged as decentralized approaches. Next, the section describes these research areas in their chronological order.

2.1.1. Cluster Computing

A compute *cluster* [53, 79, 205] consists of a large set of compute nodes and one master node [184, p. 17 f]. All components are built from off-the-shelf hardware, which makes the cluster more affordable compared to supercomputers.

The entities of a cluster are linked via a high-speed network to exchange data and execution information. Clusters are rather homogeneous in terms of their hardware and are tightly coupled. Thus, to improve the execution performance, software is tailored to the characteristics of the specific cluster it is executed on. To execute applications, the user submits jobs via the master node, similar to the batch computing paradigm. The master node controls all running jobs and their workflows. Different examples for clusters exist. One well-known cluster is the *Beowulf* cluster [17], which is Linux-based. Beowulf incorporates a middleware to maintain a tightly coupled cluster. A more lightweight approach was developed by Engelmann *et al.* [64], where the software stack on the compute nodes is reduced to a minimum to increase the computational performance of the cluster. From the software perspective, approaches like *MapReduce* [53] increase the usability and reduce the execution times of jobs that run on a cluster. It is a programing model with an API that distributes workloads and collects results. MapReduce offers parallelization as well as mechanisms for redundancy and fault-tolerance for large-scale data-intensive applications. While MapReduce is specialized on applications that have an acyclic data flow model, *Spark* [205] by Zaharia *et al.* focuses on applications, which use data sets across multiple parallel operations. Other approaches focus on the energy efficiency of job execution on cluster systems or fault tolerance in cluster environments [204]. A comprehensive survey on energy efficient cluster computing can be found in [183].

2.1.2. Grid Computing

In *grid computing*, resources are rather loosely coupled and more heterogeneous, but still dedicated for task execution [30, 41, 49, 94, 143]. Contrarily to clusters, no assumptions are made regarding their homogeneity. This implies that grid nodes have various hardware architectures, operating systems, network connections, security policies, and administrative domains. Therefore, the grid system supports mechanisms that overcome these heterogeneities and provide the access to remote resources from different administrative domains [184, p. 25 ff]. Different types of grids exist: computational grids, access grids, data grids, and data-centric grids [150]. In focus of this thesis are computational grids.

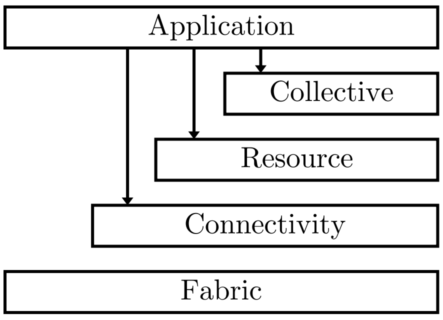


Figure 2.1.: Grid Computing Architecture by Foster *et al* [74].

Foster *et al.* defined the "grid problem", which describes the sharing of resources in so called virtual organizations that federate multiple systems and are collaborations of multiple institutions [73, 74]. As a solution for the grid problem, Foster *et al.* proposed a grid architecture that is based on the hour glass model of IP and is shown in Figure 2.1. It has five layers: the *fabric*, *connectivity*, *resource*, *collective*, and *application* layer. The *fabric layer* abstracts functionalities from the physical resources in the system and implements the local, resource specific operation. Communication protocols in the *connectivity layer* facilitate the exchange of data between fabric resources based on the TCP/IP stack. On this layer, network protocols, transport protocols, and application protocols, such as IP, ICMP, TCP, UDP, DNS, and OSPF, are used. The *resource layer* provides sharing operations on individual resources. Based on communication and authentication protocols, it provides secure mechanisms for negotiation, monitoring, and accounting. Together with the communication layer, the resource layer represents the neck of the hour glass model. So far, the layers are focused on individual interactions. The *collective layer* facilitates the collaboration of multiple resources. It considers global states, manages workload, and schedules tasks. The *application layer* is on top of the architecture. The layers below provide APIs that are used by the application to perform the desired action. Comprehensive surveys on grid computing can be found in [114, 129, 203].

2.1.3. Volunteer Computing

Volunteer or desktop grid computing systems aim to harvest idle resources of desktop computers [7, 9, 35, 132]. These resources are located at the edge of the Internet and contribute their computational capability to high throughput applications. Compared to grid computing, volunteer or desktop grid computing does not only consist of more heterogeneous and loosely coupled nodes, but also integrates end-user devices [44]. Hence, the computing nodes are less reliable, have changing network connections, and fluctuate. Resource providers in desktop grids are individually administrated by their users and not dedicated for execution. Therefore, the user can shut them down or cancel task executions at every point in time leading to a high resource volatility [45]. Resource providers can also behave maliciously, consequently reducing the execution quality of the overall system. Desktop grid systems have to cope with these characteristics by means of fault-tolerance mechanisms.

From the application perspective, standard grid applications often have dependencies between tasks and are focused on a high system performance. In contrast to that, desktop grid applications have no dependencies between tasks and are focused on high throughput. According to Choi *et al.* [45], volatility, dynamic environments, lack of trust, failures, heterogeneity, scalability, and voluntary participation are the main challenges in desktop grid computing. These challenges have an impact on resource management and scheduling in desktop grids. Prominent systems like BOINC [5], Condor [132, 179], and Entropia [35, 43] cope with these challenges by means of fault tolerance mechanisms and specifically designed scheduling algorithms. A survey on volunteer computing can be found in [44].

2.1.4. Cloud Computing

Cloud computing is based on the idea of utility computing, which was mentioned by John McCarthy in 1961 at the MIT's centennial celebration. In this vision, computing is a public utility similar to the telephone system. Users need to pay only for the resources that they actually use. Today, different cloud providers offer services that fulfill McCarthy's idea (e.g., Microsoft Azure[1], Amazon Web Services[2], and Google App Engine[3]). The national institute of standard and technology (NIST) defines cloud computing as follows [140]: *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of*

[1]Microsoft Azure: https://azure.microsoft.com/, accessed: 10/01/2019
[2]Amazon Web Services: https://aws.amazon.com/, accessed: 10/01/2019
[3]Google App Engine: https://appengine.google.com/, accessed: 10/01/2019

configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

In the year 2006, Amazon made cloud computing popular and offered parts of their infrastructure as a service. Based on the economy of scale, they deployed large global data centers and linked them to the Internet via high bandwidth connections. Originally, Amazon built the infrastructure for their own use to host their marketplace. They observed that parts of their infrastructure are unused most of the time. This over-provisioning is expensive, but necessary to keep up with performance peaks. Amazon started renting these excess capacities to customers.

From the customer perspective, cloud computing provides high flexibility based on the pay-as-you-go principle. It offers a pool of various IT resources to the user that are scalable, simple to use, and centrally administered, without the need of up-front investments [82]. Using 1000 servers for one hour leads to the same cost as using one server for 1000 hours. This established new possibilities for companies that make use of data analytical batch jobs [10]. In [76], Foster *et al.* compare cloud and grid computing comprehensively. The result shows that the visions, architectures, and technologies are similar. However, in terms of security, programming model, business model, compute model, data model, application, and abstractions, grid and cloud systems differ.

Cloud computing introduces three different service levels: infrastructure as a service, platform as a service, and software as a service [140]. Zhang *et al.* [207] further introduced a more comprehensive perspective on cloud computing, shown in Figure 2.2. On the bottom, the hardware layer consists of physical servers, routers, switches, power, and cooling systems. The infrastructure layer sits on top of the hardware and realizes virtualization of data centers. With virtualization technology like XEN [15] or vManage [118], physical hardware resources can be partitioned into virtual machines. The infrastructure layer allocates and manages the cloud resources to tailor virtual machines to the specifications of the cloud customers. Examples for infrastructures as a service on that level are Amazon Elastic Compute Cloud[4] (EC2) and Flexiscale[5]. The third layer facilitates

[4]Amazon EC2: https://aws.amazon.com/de/ec2/, accessed: 10/01/2019
[5]FlexiScale: http://www.flexiscale.com/, accessed: 10/01/2019

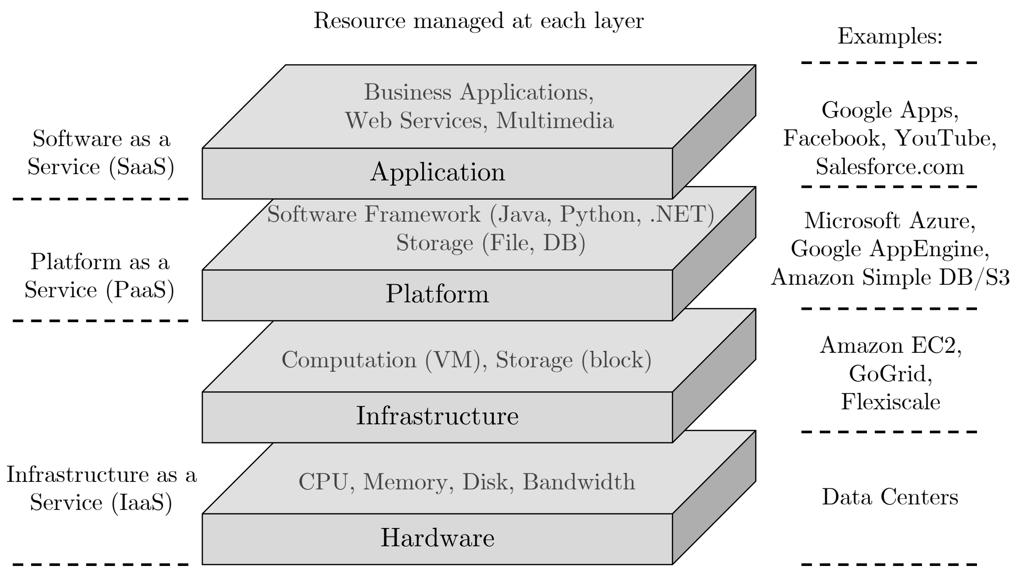

Resource managed at each layer
Examples:
Business Applications, Web Services, Multimedia
Application
Software as a Service (SaaS)
Google Apps, Facebook, YouTube, Salesforce.com
Software Framework (Java, Python, .NET) Storage (File, DB)
Platform
Platform as a Service (PaaS)
Microsoft Azure, Google AppEngine, Amazon Simple DB/S3
Computation (VM), Storage (block)
Infrastructure
Amazon EC2, GoGrid, Flexiscale
CPU, Memory, Disk, Bandwidth
Hardware
Infrastructure as a Service (IaaS)
Data Centers

Figure 2.2.: Cloud computing architecture by Zhang *et al* [207]. The different layers and their respective abstraction level are presented. On the left and right side the service levels and examples for services are shown, respectively.

platform as a service based on operation systems and application frameworks. This higher level abstraction offers an easy development and fast deployment of cloud applications. As an example, the Google App Engine provides an API to implement all elements of a typical web application. On top of the architecture, the application layer runs the cloud application, which can be scaled automatically in contrast to standard applications. This layer provides software as a service, such as Salesforce.com[6], which offers cloud-based customer relationship management solutions.

In general, cloud computing provides several service types. Amazon Web Services provides more than 90 different services, such as compute, storage, artificial intelligence, and IoT services. All of these services can be integrated mutually to tailor a specific cloud application. Especially compute services are in the focus of this thesis. Besides the EC2 that offers customized virtual machines, Amazon Lambda provides the execution of single methods without provisioning or managing cloud servers. Other services facilitate auto-scaling of cloud applications

[6]Salesforce.com: https://www.salesforce.com/, accessed: 10/01/2019

depending on the workloads. Since 2006, cloud computing faced many challenges regarding privacy, security, latencies, and costs [100, 155, 178]. Surveys on cloud computing can be found in [34, 153].

2.1.5. Mobile Cloud Computing

The term *mobile cloud computing* (MCC) is used for different types of systems and describes a special form of cloud computing with mobile devices as resource consumers [57]. Based on the MCC idea, resource-scarce mobile devices can augment their computational capabilities with cloud resources and, for example, offload computationally intensive tasks. The challenges of these systems are the limited network connection, heterogeneity, device mobility, and system security. Recently, the term MCC is also used for approaches that deploy smaller cloud servers at the edge of the network in proximity to the users like Cloudlets [158, 159, 160]. Other approaches define so called *mobile ad-hoc clouds* that consist of as set of close-by mobile end-user devices. In case these devices agree on sharing their resources, they join a nearby group and exchange computational capabilities. Yaqoob *et al.* present a survey in [199].

2.1.6. Fog Computing

The *fog computing* paradigm was introduced by Cisco [24] to extend cloud computing with resources deployed at the edge of the Internet. This is based on the emerge of the IoT, which introduced new requirements for computational resources, such as mobility support, location-awareness, and lower latencies [23]. Fog computing is defined as *"a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of network"*. Based on that, fog computing works in conjunction with cloud computing, pairing the strength from both areas. This leads to a system with high performant cloud and low latency fog resources. Gradually, the fog paradigm became more and more independent from cloud computing, which led to a high performance heterogeneity amongst the fog nodes [154]. As a result, a three-tier architecture with clients, fog nodes, and central cloud servers [134] arose, offering new opportunities for fog computing. This architecture can be applied in different application areas such as

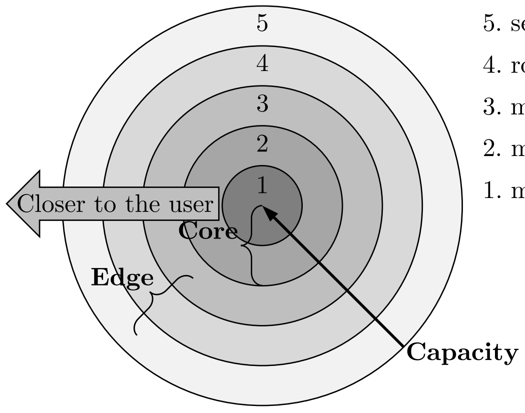

5
4
3
2
1
Closer to the user
Core
Edge
Capacity
5. sensors / meters /mobile phones / cameras
4. routers / gateways / set-top boxes / access-points
3. modular data center / container / local cluster
2. modestly-sized data center (distributed cloud)
1. mega data center (large centralized cloud)

Figure 2.3.: From the cloud to the edge by Li *et al* [128].

augmented reality [85], cyber-physical systems [176], and application in vehicle to vehicle and vehicle to infrastructure communication [109]. Comprehensive surveys on fog computing approaches can be found in [137, 201, 202].

2.1.7. Edge Computing

The term *edge computing* – as it is used in this thesis – was first mentioned in 2014 by Vaquero *et al.* [185] and became a standalone paradigm for decentralized computing. Edge and fog computing are used interchangeably by some authors [42, 170]. In this thesis, they are handled separately with different emphases. In a nutshell, edge computing is more decentralized and more focused on the collaboration of user-controlled devices than fog computing.

While cloud resources offer enormous capacities, they entail fundamental problems. According to Lopez *et al.* [77] these problems are: the loss of privacy and social data, the delegation of application and service control, and the large amount of communication overhead. As a solution, the edge computing paradigm deducts applications, data, and services from central cloud resources and deploys them on user-centric edge devices. The approach retains the cloud as a central support infrastructure for a few number of tasks. Figure 2.3 shows the edge-oriented perspective on the Internet based on Li *et al.* [128]. In the center is the core that is based on clouds and data centers. On the next layer, smaller web servers are located. At the edge of the Internet, user-controlled devices such as PCs, mobile phones, and sensors take over the majority of tasks. Their advantages encompass proximity, intelligence, trust, control, and human-centered applications at the edge [77]. Comprehensive surveys on edge computing can be found in [128, 190].

2.1.8. Mobile Edge Computing

IBM and Nokia first mentioned *mobile edge computing* (MEC) in 2013 when they introduced a platform to run applications at mobile network base stations. One year after that, the European Telecommunications Standard Institute (ETSI) launched a MEC group [148] to create a new standard. The general idea of MEC is to deploy infrastructure with cloud capabilities at the edge of the mobile network [154]. This infrastructure can be deployed at different locations in the Radio Access Network, such as the LTE/5G base stations, 3G radio network controllers, or multi-radio access technology call aggregation sites. Mobile network operators are responsible for the installation and maintenance of mobile edge hardware. Based on the ETSI white paper by Patel *et al.* [148], the main characteristics of MEC are as follows: (1) *On-premises:* the mobile edge is local and can be used if isolated from the rest of the network. (2) *Proximity:* it is close to the source of information to support computationally intensive applications. (3) *Lower latency:* due to the device proximity, low latencies and high bandwidths contribute to a high user experience. (4) *Location awareness:* edge devices use low level signaling to obtain location information of the participating devices. (5) *Network context information:* applications and services can use real-time information about the network to offer better services. In [2], Abdelwahab *et al.* apply the mobile edge paradigm and design a solution based on LTE networks. More recently, Tran *et al.* [181] published their vision of a MEC framework in the 5G network. Based on that, new technologies can be supported such as IoT, augmented reality, and connected cars [95]. Comprehensive surveys on MEC can be found in [1, 136, 138, 190].

2.2. Computation Placement and Offloading

In this thesis, computation placement and computation offloading are used as separate terms. The latter depicts the process of extracting a computationally intensive part of an application and transferring it to a remote resource for execution. The result is sent back to the calling application afterwards. Computation placement, on the other hand, adds further abstractions to offloading. It decides on a distribution strategy for application intensive parts and splits them up such that resources are used optimally. Computation placement encompasses performance-aware tailoring of workloads as well as runtime migration of application parts to facilitate the required level of resource elasticity in the system.

The device intending to place computation remotely is referred to as the *local device*. Computation placement can have multiple reasons such as, improving scalability, saving energy, migrating code transparently, and optimizing responsiveness. Nevertheless, it entails challenges that need to be considered to exploit these benefits. Flores *et al.* [69] proposed a generic offloading architecture and structured their analysis on the following questions: *what*, *when*, *where*, and *how* to offload to obtain a time and/or energy benefit for the local device. Kumar *et al.* have a similar model for their literature analysis in [116]. Based on that, the rest of the section is structured by means of these questions to examine the characteristics of computation placement and offloading.

What to Offload?

The question of *what to offload* refers to parts of code of the consumer application. A computationally intensive part of an application is selected for offloading, if a remote execution saves time, energy, or both. Depending on the approach [116], the level of granularity of these parts may vary from methods [8, 90], tasks [36, 65], applications [43, 66], to entire virtual machines [113]. The identification of these parts, however, is not trivial. Some approaches, such as MAUI [48] or ThinkAir [112] use annotations in the source code. These annotations are provided by the developer during the application development phase. This rather manual approach assumes knowledge about the runtime behavior and the complexity of the program. This assumption does not always hold. Therefore, other approaches apply a different strategy that is rather dynamic in term of *what to offload*. These approaches make the offloading decisions based on static code analysis or history traces. The component that decides on *what to offload* is called a *code profiler* [69].

When to Offload?

When to offload refers to the decision if a part that is selected by the code profiler is actually offloaded or not. This decision depends on the current situation of the local device and is made by the so called *decision engine* [69]. The decisions can be made statically or dynamically. In the static case, the parts that were selected by the profiler are always remotely executed, regardless of the current context. This may lead to bad responsiveness or even higher energy consumption. Therefore, most computation placement systems monitor the local devices in terms of their performance, battery status, and network connection. Based on that and the required data transmission size, approaches decide if offloading is beneficial or not. They define the benefit of offloading depending on computation and communication effort [117, 116]. If the time for a local execution is longer than the remote execution plus the data transmission time, offloading is beneficial in terms of performance. Energy saving benefits are computed analogously. If the local execution consumes more energy compared to the remote execution plus the energy for data transmission, offloading is favored. Therefore, local execution is the default and code offloading is optional and only done if beneficial.

So far, this section assumes offloading systems that employ stable cloud resources as providers. With unstable and fluctuating edge resources, the *when* question is far more complex to answer. In case of plenty powerful edge resources with good network connections, computation placement is potentially more beneficial compared to a situation with only a few unreliable devices in the environment. Thus, the *when* question is influenced by the *where* question.

Where to Offload?

Where to offload refers to the remote resource provider that executes the offloaded code. This location can again be static and dynamic. In case of stable cloud resources or other dedicated servers, the approach is static. Dynamic placement decisions in terms of resource providers are more complex, since context information of devices in the environment must be considered. In computation placement systems that utilize edge resources this has a major influence, since these resources are user-controlled. The current amount of available resources and their stability also determines if remote computation placement is even beneficial or not. To collect context information, all devices in the environment are monitored. The gathered information is used to make the scheduling decisions. The system component that monitors the environment is called *system profiler* [69].

How to Offload?

How to offload refers to the remote executions mechanism. The so called *surrogate platform* is the remote execution instance in a computation placement scenario [69]. It recreates the state and runtime environment of the consumer device. Therefore, it is able to execute the methods, tasks, application, or virtual machines in the same way the local device does. Depending on the granularity and the level of abstraction, this is achieved by remote procedure calls (RPC), bytecode transmissions, virtualization, or even by running clones of entire virtual machines. The parallel execution of offloaded tasks is also achieved by the surrogate platform. This can be realized through different mechanisms that provide redundant task scheduling or task partitioning.

2.3. Pervasive Computing and Context-aware Computing

Especially in pervasive systems and the IoT where thin and resource scarce devices are common, the potential benefit of the proposed approach is substantially. Moreover, the benefit of offloading from edge devices is largely determined by their current context. Therefore, context, context-awareness, and IoT are relevant for this thesis and are introduced next.

In his article 'The Computer for the 21st Century' [191] that he published in 1991, Mark Weiser laid the foundation for modern pervasive and ubiquitous computing systems, including the IoT. In his vision, computers and other interconnected computational devices become interwoven with objects of our everyday life and at some point in time vanish into the background entirely. To realize his vision, these systems require context-awareness. The concept of context and context-awareness was defined in 1994 by Schilit *et al.* [168]. According to them, context has three key features, which are: (i) the user's location, (ii) the user's social group at the same location, and (iii) the nearby resources. The perspective on context changed over time and authors like Schmidt *et al.* proposed new working models for context that are focused on sensor-based context-aware applications. The most prominent and general definition of context is given by Dey [56]: *"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*

Similar to context, the definition of context-aware computing evolved steadily over time. In 1999, Dey and Abowd published their perspective on context-aware computing: *"A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."* They further described the three categories of features that a context-aware application can support [4]: *"(i) presentation of information and services to a user, (ii) automatic execution of a service for a user, and (iii) tagging of context to information to support later retrieval."*

The *Internet of Things* is a modern paradigm that is based on this foundation. According to Atzori *et al.* [11], IoT consists of three visions: the *Things*-oriented vision, the *Internet*-oriented vision, and the *Semantic*-oriented vision. The first vision refers to things which are everyday objects equipped with computing hardware, such as RFID tags, sensors, or actuators. The Internet-oriented vision describes the networking aspect of the paradigm. It includes specialized versions of IP for smart objects or the Internet as a general medium of communication. The semantic-oriented vision refers to any data reasoning and semantic technology that makes it possible to control the huge amount of integrated elements. IoT is closely related to the edge computing paradigm, since application in this domain rely on low latency computing resources [198]. Hence, IoT is a key enabler of fog and edge computing [92][7].

This chapter discussed the background of this thesis with respect to distributed computing systems, computation placement and offloading, as well as pervasive and context-aware systems. The following chapter introduces a scenario and derives requirements for the design of this thesis.

[7] [92] is joint work with M. Heck, J. Edinger, and C. Becker

3. Requirement Analysis

This chapter conducts a requirement analysis based on the research questions of this thesis. Therefore, a scenario is introduced that describes all players and system entities as well as their interactions. Based on that, the functional and non-functional requirements for the design are derived in Section 3.2 and 3.3. These requirements are the foundation for the design and prototype implementation.

3.1. Scenario

In the modern computing landscape various devices exist that have computational excess capacities. Thus, a vast amount of computational power is unused. This capacity is contributed to a global distributed computing system. These devices range from standard user-controlled devices like desktop PCs, smartphones, and tablets to specialized hardware solutions, such as in cloud and grid computing environments. As a solution, a middleware serves as an abstraction on top of this computing landscape and hides the distribution complexities from the application developer. It abstracts the plain computational capabilities from the otherwise heterogeneous devices owned by resource providers. This middleware offers an easy-to-use API allowing the integration of generic applications from all kinds of domains. Artificial intelligence, virtual reality, and machine learning, are only a few of recent trends that require a large amount of resources. These applications can benefit from offloading computation via the middleware.

To do that, the developers first identify the computationally intensive parts of the application. They replace these parts with API calls to the middleware in order to initiate the remote execution. To tailor the task execution, developers specify the required quality of service level, in terms of, for example, reliability, speed, or responsiveness. Additionally, they specify how a task can be partitioned, so that the system can exploit parallel execution of a task. Depending on the application requirements, the middleware transparently enforces the task execution

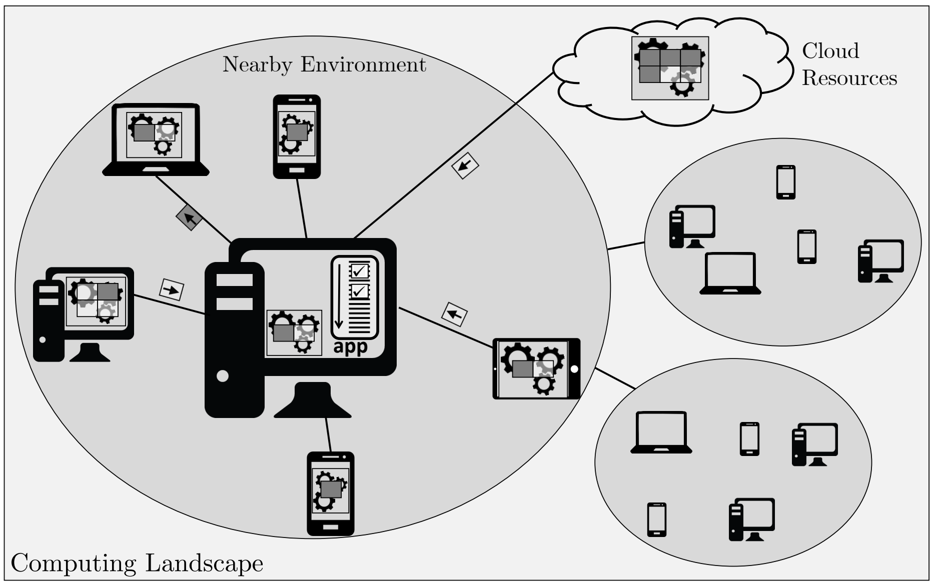


Figure 3.1.: In this scenario, computation is seamlessly exchanged between applications and any local or remote edge resource that contributes to the computation placement system. The resource intensive parts of an application are allocated on idle resource providers. These remote resources execute the offloaded tasks (*dark gray*) and return the results (*light gray*) back to the application. These resource providers can be in the proximity of the resource consumer to provide low latencies.

by scheduling on suitable resource providers. When scheduling on unreliable edge resources, the middleware provides robustness by ensuring that the task is eventually executed. In case that the application requires high performance resources for long-running tasks, the middleware may schedule the execution on cloud resources or on remote edge resources with high performance. For applications that need high responsiveness or security, resources in proximity are used for execution. In many cases, applications demand a high execution performance and responsiveness. To facilitate both, the middleware can increase the responsiveness of cloud resources or increase the performance of edge resources. The first approach integrates cloud resources into the nearby environment of the user – known as fog computing – to reduce response times. The second approach employs edge resource more efficiently to increase their performance. The scheduling mechanism ensures elasticity in the edge by means of an optimal workload distribution and resource provisioning. It also decides on the optimal strategy for the application. After the execution on the resource provider, the result is sent back to the application. The scenario is illustrated in Figure 3.1.

The system model for this scenario assumes the participation of the majority of the existing edge devices. Further, these devices must be able to communicate with each other via a network. From the consumer application perspective, it is assumed that the computationally intensive parts of an application are identified by application developers.

3.2. Functional Requirements

From the scenario, a set of seven functional requirements is derived in this section. These requirements are relevant to the approach that is designed and implemented in this thesis, which is referred to as *"the system"*.

$\mathbf{REQ_{F1}}$ Computation Placement: The major purpose of the system is to place computation on remote resource providers. This requires a well-defined API that allows to specify computationally intensive parts of an application. Depending on the context, the system should decide on the most suitable resource provider and provide means to tailor the execution behavior of tasks. Some applications demand a specific set of execution requirements to run successfully, such as high performance, responsiveness, or reliability. The system shall consider these requirements and allow the developer to tailor the execution accordingly. Finally, the system must deliver the computation result to the application according to application requirements.

$\mathbf{REQ_{F2}}$ Lightweight Computation Abstraction: The system requires a computation abstraction to represent the computational logic of a task and to enable its execution on any device in the system. It must be lightweight to keep the communication costs reasonable and to allow small devices the execution.

$\mathbf{REQ_{F3}}$ Edge Support: The integration of edge resources is a major requirement of this approach. The system shall abstract the capabilities of edge resources to approximate the execution quality of cloud resources without their drawbacks. Therefore, the nature of edge resource must be considered, namely, fluctuation, unreliability, and locality. The system shall cope with these shortcomings and fully exploit the benefits of these resources, especially locality. Due to their significance, edge elasticity and heterogeneity are handled separately.

REQ$_{\mathbf{F4}}$ Edge Resource Elasticity: The elasticity of resources in the edge is required to provide the system with a sufficient set of resources. In cloud computing, the ad-hoc provision of resources and the adaptation to the current resource demand is called elasticity [93]. The system shall adapt the provisioning of edge resources to the demand of the user application and the current workload of the system. This assumes that a certain amount of users participate in the system and contribute their resources.

REQ$_{\mathbf{F5}}$ Overcoming Edge Heterogeneity: Especially at the edge, heterogeneities come in different forms such as hardware architecture, operating system, programming language, accessibility, and task characteristics [166][1]. To achieve the vision of computation as a common good, the heterogeneity of the computing landscape must be tackled by the system. This includes the interoperability of all participating devices, regardless whether the devices have different operating systems, hardware architectures, network connections, or computational performances. Besides, heterogeneities of consumer applications such as programming language and specific execution requirements must be considered as well.

REQ$_{\mathbf{F6}}$ Hiding Complexities: The process of computation placement entails major complexities. The system should hide these complexities from all system participants. The API of the system should support the application programmer with well-defined methods to determine the required execution quality on an abstract level. Further, resource consumers should not be aware of the system executing tasks remotely. Thus, the system shall include access, location, migration, replication, and failure transparencies.

REQ$_{\mathbf{F7}}$ Unobtrusive: On the resource provider side, the execution of tasks shall not interfere with the local user. The system should monitor the local usage and adjust the amount of task execution accordingly. This guarantees that the local user should not be aware of the execution of tasks for resource consumers.

[1][166] is joint work with J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker

3.3. Non-Functional Requirements

In addition to the functional requirements, the system must fulfill a set of four non-functional requirements.

REQ_{NF1} Performance: The system performance is a key factor for computation placement. It can be measured by different metrics like reduced energy consumption, response time, and network cost. At least one of these factors must be positive to decide for a remote task execution. The system requires a performance measure which is focused on the response time and data overhead. It neglects the energy consumption of devices.

REQ_{NF2} Scalability: Neumann defined three dimensions for scalability of distributed systems [144]: numerical scalability, geographical scalability, and administrative scalability. Numerical scalability refers to the number of users, meaning, that by adding more users and resources, the system does not decrease its performance. The second type of scalability targets functionality of the system despite the geographical distance between the participants and the arising communication delays. Administrative scalability refers to a system that spans over several independent organizations and is still manageable. The proposed system shall considers all three dimensions of scalability and be able to manage an unlimited amount of resources from various organizations that are globally distributed.

REQ_{NF3} Robustness: The system must cope with errors, failures, and malicious behavior of participating nodes. Especially in edge computing environments where end-user devices are used as resource providers, malicious behavior is likely. The system shall apply mechanisms to increase the execution qualities of edge resources despite of their unreliable nature. These mechanisms are transparent for the user and the application developer.

REQ_{NF4} Extensibility: The system should be extensible in terms of hardware architectures, application requirements, and mechanisms for robustness. Due to the fast evolution of the computational landscape, an easy extension and replacement of software components is important for the research design.

After the requirements analysis, related approaches are investigated in the next chapter.

4. Related Work

This chapter presents the related work of the thesis. First, a classification for the literature analysis is developed. Second, the different related areas are examined by presenting the most prominent approaches. Third, the literature analysis is summarized by means of an overall classification of the approaches and the identification of the research gap.

4.1. Classification

The classification reflects the requirements of the previous chapter. Four general classes emerge from that: *heterogeneity*, *edge support*, *elasticity*, and *usability*. These classes are divided into subclasses. Heterogeneity consists of *operating system*, *hardware architecture*, *programming language*, *accessibility*, and *task*. Especially accessibility and task heterogeneity require explanation. Accessibility includes different network technologies as well as bandwidths and latencies in the same system. Task heterogeneity means the irregularity of tasks considering their computational effort. The classification of heterogeneity is based on [167][1], where further detail can be found. The second class is edge support, which describes the general ability of using edge devices as resources, as well as to cope with device *churn* and *mobility*. The third class is elasticity which has two subclasses: *adaptability* and *efficiency*. Adaptability describes to which degree a system can adjust to the current workload demand of the applications. This is influenced by the efficiency, which determines, how well a system can exploit the existing resources. The last class is usability, which is determined by *abstraction*, *transparency*, and *obtrusiveness*. The first subclass determines if a system offers a computation abstraction for the application developer. In order to do that, systems may offer certain programming models or entire frameworks. The transparency of a system, which is also relevant for the application developer,

[1][167] is joint work with J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker

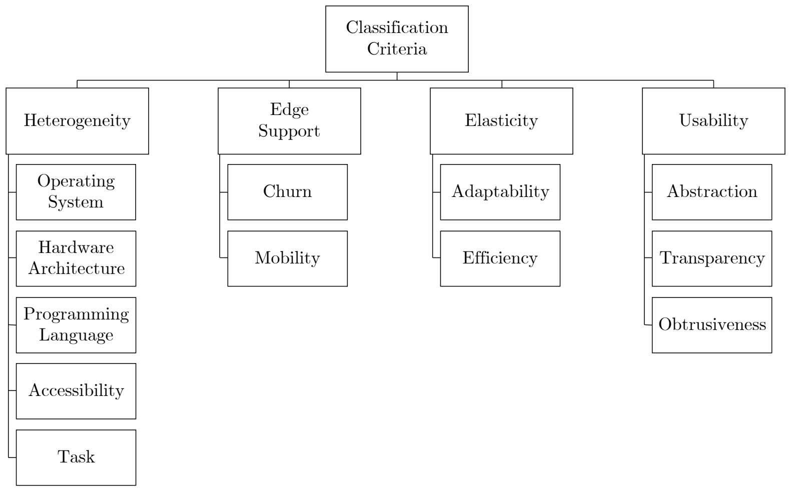


Figure 4.1.: Overview of the literature classification approach.

is determined by the level of complexity that is hidden. This encompasses access, location, migration, replication, and failure transparencies. Obtrusiveness is a relevant usability factor for the providers who share their resources. It is defined by the degree of perceptible interference that is implicated by the task execution for other participants. Figure 4.1 shows an overview of the classification.

4.2. Literature Analysis

After the classification, the relevant approaches are presented in this chapter. In general, the literature evaluation encompasses approaches with different scopes. Some are comprehensive and tackle various challenges of distributed computing systems while others are solutions to rather specific problems. In both cases, the approaches are classified to determine their relation to the thesis.

The literature review does not claim to be exhaustive, since the relevant period ranges over three decades and encompasses several areas of distributed computing systems: First, cluster and grid computing systems are described and classified. Second, the chapter analyzes volunteer computing approaches. Third, cloud and MCC approaches are presented and categorized. Fourth, the area of edge computing is examined. It consists of fog computing, mobile data clouds, and hybrid approaches. Lastly, computation offloading systems are discussed.

4.2.1. Cluster and Grid Approaches

The resources in cluster and grid computing are more stable, more homogeneous, and closer-coupled compared to edge computing. Nevertheless, research in these areas laid the foundation for computation placement systems in terms of resource management (e.g.[19, 30, 41, 49]), overcoming heterogeneity (e.g.[182]), and workload balancing (e.g.[41, 129]). Other approaches deal with trust [12, 197], economy [31, 33, 194], virtualization [67, 115, 174], energy-awareness [79, 183] or data management [94, 111]. In the following, cluster and grid approaches are presented.

MapReduce [52, 53] by Dean and Ghemawat is a programming model as well as an implementation for processing large data sets on clusters. The computation abstraction for the programmer is handled with the *map* and the *reduce* function, both written by the user. The map function takes an input key/value pair and generates an intermediate key/value pair. The reduce function takes intermediate key/value pairs with the same key and produces zero or one output values. Thus, with MapReduce, the programmer splits up the job into the smallest granularity possible and, after that, the system automatically handles the parallelization. This includes tasks, such as partitioning, allocation and scheduling, fault-handling, and communication among nodes. MapReduce is, however, limited to a certain set of applications, in particular those using a working set only once.

For many applications from the machine learning and graph algorithm domains, this limitation is a problem. As a solution, Spark [205] by Zaharia *et al.*, is focused on applications that reuse working sets across multiple parallel operations. Spark uses so called resilient distributed datasets which are read-only collections of objects. These objects can be used across different parallel operations. Most early cluster approaches assume that the nodes in a compute cluster are homogeneous, which is rarely the case. Over time, Zaharia *et al.* integrated several operating systems. Further, they tackled performance heterogeneity in [206] to reduce the loss of computational power. In order to do that, the authors present the LATE (*longest approximated time to end*) scheduling algorithm that is robust to heterogeneity. It, however, only operates under the assumption that the progress of a task is linear to the completion time.

The Dryad approach [99], presented by Isard *et al.*, has a similar goals as MapReduce, but achieves them with a different design. It supports coarse-grained applications and has a graph-based representation, where vertices are computation and edges are communication channels. A job manager handles the execution of the graph. In case all inputs of a vertex are available, it is runnable and can be executed. While the map function of MapReduce can only take one input, each vertex of Dryad can have multiple inputs.

In [108], Kim *et al.* presented SnuCL, an OpenCL-based framework for heterogeneous CPU/GPU clusters. This framework exploits closely-coupled clusters that consist of CPU and GPU computing hardware. With SnuCL, standard OpenCL programs can be deployed and distributed on mid-size clusters transparently for application programmers. Similar to that, LibWater [81] by Grasso *et al.* extends OpenCL for heterogeneous clusters as well. In contrast to SnuCL, LibWater offers further abstraction and eases the programmability in terms of data transfer and MPI handling. The LibWater runtime system optimizes the efficiency of programs transparently and is more scalable.

Next, grid approaches are dicussed. The Globus project [72] by Foster and Kesselman focuses on the configuration and performance optimization of metacomputer environments. In their definition, a metacomputer is a networked virtual supercomputer that is dynamically built from distributed computing resources, also known as grid. Globus enables the modular deployment of a grid system by providing different kinds of services, such as security, resource management, data management, and communication. With these services, it forms an adaptive wide-area resource environment. Further, Globus adds a quality of service (QoS) component [75] that is based on resource reservation and application adaptation. Buyya *et al.* introduced Nimrod/G [30] as an extension of the Globus middleware. Nimrod/G is a resource broker and focused on the management and scheduling of computation within a grid environment. Further, it adds an computational economy and introduces a market-based model for resource management.

The problem of many early grid approaches is the barrier of users to participate in a system. The OurGrid approach [7] by Andrade *et al.* is an easy-to-access, open, and extensible grid platform. It is based on a network of favors, meaning, that users who contribute a large part of their resources are prioritized when they request resources from others. OurGrid is focused on so called bag of task (BoT) applications. A BoT application consists of a set of independent tasks that are not required to communicate with each other during execution. These applications are suitable to be executed in a grid environment.

The SpeQuloS approach [54, 55] by Delamare *et al.* combines grid and cloud. They identify bottlenecks as one of the main problems of BoT executions on best effort grids. As a solution, SpeQuloS improves the QoS for these applications in three ways: first, it reduces the completion time. Second, it improves the execution stability, and third, it informs the user about a statistical prediction of the BoT completion time. To do so, SpeQuloS detects potential bottlenecks and counteracts with scheduling critical tasks on reliable cloud resources. It monitors the task execution and uses different resource provisioning strategies for the cloud. In case of an execution on cloud resources, SpeQuloS checks the accountability of the user and predicts a task completion time.

Gridbot [172] by Silberstein *et al.* creates a single virtualized computing platform from multiple grids, cluster, and volunteer environments. The approach facilitates the execution of a BoT application across different grids concurrently. Gridbot unifies all grid infrastructures by establishing an overlay of resource providers. It focuses on a rapid turnaround time by resource allocation mechanisms, task replication, and dynamic bundling of tasks for the same grid type. Gridbot encompasses prioritization policies to execute multiple BoTs concurrently. The implementation of Gridbot is based on the BOINC framework.

Compared to clusters, grids are less reliable, which is tackled by grid-specific fault tolerance approaches, such as [83, 87, 97, 101, 106, 173]. These approaches use task replication strategies or a proactive fault prediction to increase the reliability and the throughput of the grid. In [180], Townend and Xu construct a failure model for grids and identify timing, omission, and interaction faults as prevalent. They cope with failures and malicious behavior by applying replication and majority voting mechanisms. A similar approach [131] by Litke *et al.* uses static replication, meaning, the number of replications is not changed in case of failures. In [156], Rood and Lewis increase the efficiency of replication by predicting the likelihood of a successful job execution. Thus, jobs are only replicated in the grid if necessary. Hence, they drastically reduce the number of replications while retaining the success rate of execution nearly stable. Likewise, the approach by Gurun *et al.*, presented in [84], uses a Bayesian approach for the prediction of the reliability. Another similar approach by Huda *et al.* [96] analyzes the different failure sources in a grid, namely hardware, application, operating system, network, software, response, and timeout faults. They use a so called schedule advisor to prepare a recommendation for the execution plan. In [78], Garg and Singh discuss checkpointing and migrating in combination with replication of tasks to increase the fault tolerance in a grid environment.

Other approaches in grid computing optimize the dispatching and scheduling of tasks (e.g. [59, 88, 123, 127, 193]). Falkon [151], uses a multi-level scheduler that separates resource allocation and task dispatching and exploits bundling of tasks to reduce communication effort. Xhafa and Abraham [195] define the grid scheduling problem and present different heuristics and meta-heuristics as a solution. In [124], Legrand and Touati analyze the behavior of multiple independent and non-cooperative application-level schedulers in a single grid system while executing BoT applications. As a result, they show that cooperation increases the overall efficiency of the grid dramatically.

4.2.2. Volunteer Computing Systems

In general, this thesis is strongly related to volunteer computing, since end-user devices are used as offloading resources. However, most volunteer computing approaches are static in terms of resource integration [40], application support [5, 40, 43], or resource characteristics [5, 187].

HTCondor [132, 179] was one of the first desktop grid approaches and initially introduced by Litzkow *et al.* in 1988. HTCondor gathers idle workstations in so called resource pools and allows to share computation amongst the participants. It provides a remote system call mechanism to preserve the local environment of the consumer application on each resource provider. The so called collectors receive and store service advertisements from all participating resources in the system. Based on the classified advertisement language, resource requests can be formulated and send to the centralized scheduler system that answers with a set of potential resource for execution. During the execution, checkpointing and migration mechanisms support the fault tolerance of the system. However, HTCondor is limited in terms of QoS measures as well as security, unobtrusiveness, and protection from malicious applications.

The Spawn approach [187] by Waldspurger *et al.* harnesses idle computation time of workstations similar to HTCondor, but with further contributions. It creates a computational economy based on the otherwise idle computing resources. Participants in the system can be resource buyers as well as sellers, both attending auctions to buy and sell idle computing times. Compared to HTCondor, Spawn provides fair dynamic load sharing and supports resource management for concurrent computations in a decentralized fashion.

The Berkeley Open Infrastructure for Network Computing (BOINC) [5] is a platform for scientific computing projects. With BOINC, scientists can create and operate volunteer computing projects. It especially supports applications with large storage and communication requirements. The participants can select the project that they want to support and contribute their computational capabilities. The design of BOINC reduces the entry barrier of research projects to exploit public volunteer resources and provides different incentives for participants. The architecture of BOINC includes scheduling servers, which allocate tasks on workers, and data servers that manage the exchange of input and output files. Redundancy mechanisms provide fault tolerance and the system scales to millions of users. Example projects are Seti@Home [6], which aims at finding extraterrestrial life forms, and Folding@Home [16], which simulates protein folding to get a better understanding of diseases such as Alzheimer's and Parkinson's. With BOINC, however, the coupling of participants and projects is rather fixed, leading to a single-application system. Further, the application structure is limited to batch jobs.

Similar to HTCondor and BOINC, XtremWeb [37, 66] builds a global-scale computing platform for scientific applications. The authors claim that the approach is more decentralized. XtremWeb uses three different types of nodes: clients, workers, and coordinators. The exchange of messages is based on RPC. XtermWeb uses sandboxing to provide data privacy and protection against malicious nodes. To increase the fault tolerance, replication and checkpointing mechanisms are added in a later approach [58]

In contrast to BOINC, Entropia [35, 43] by Chien *et al.* supports a variety of applications. Entropia offers sandboxing based on virtualization technology that runs entire processes of various programming languages in an isolated environment. By means of that, it supports security for the application as well as for the resource owner. Further, the execution of these jobs can be stopped in case the resource owner requires the performance locally. All data is encrypted and the file system API calls are mapped from the standard directory to the Entropia environment respectively. Entropia, however, only offers coarse-grained offloading of entire applications.

The Nebula system [40, 103, 157, 192] is a volunteer approach that is designed for data-intensive applications. It uses end-user devices to deploy more dispersed and less managed infrastructures compared to other approaches. With Nebula, the interaction between compute and storage nodes is closely coupled to gain location-awareness within the resource management. It employs fault tolerance mechanisms, considers performance heterogeneity of the participating nodes, and deploys the MapReduce programing model. Nebula, however, does not consider different hardware architectures, sandboxing, nor APIs for the integration of generic applications.

Device failures and fluctuation can cause large bottlenecks in volunteer computing systems. In [152], Ren *et al.* propose a failure prediction model that is based on a semi-Markov process. With this model, two types of failures are predicted with an accuracy of over 86%. The system proactively creates checkpoints of the progress and triggers a migration to another computation resource. As a result, the computation is continued on an error-free devices without any lost computation. The migration process, however, introduces a message and data transfer overhead. In the literature, several approaches [38, 39, 188] apply similar strategies for fault-tolerance.

4.2.3. Cloud Computing Systems

In this section, offloading to cloud resources is examined. Cloud computing focuses on virtualization technology and copes with virtual machine creation, management, and migration [15, 67, 120, 135, 149]. These virtual machines can be images of single applications or entire operating systems. The approaches optimize the resource provisioning to enable elasticity for offloading systems. Compared to volunteer and edge computing, offloading to cloud resources comes at higher cost. Cloud computing research is distantly related to the thesis, since it is focused on providing platforms for static infrastructures [36, 186]. Compared with that, MCC is often used for computation offloading and augments the capabilities of mobile devices and is therefore more relevant.

In [105], Justino *et al.* present a mobile offloading approach that uses the Aneka cloud computing framework to augment the computational capability of Android devices. On the Android side, they developed the so called Aneka mobile client library, which provides an easy integration of existing apps to delegate resource intensive tasks. It further connects the cloud resources and manages serialization, de-serialization, and the message exchange. On the other side, the Aneka cloud platform handles the resource provisioning, the job scheduling, and is able to encapsulate different cloud providers. The system transparently offloads computation from the mobile devices to the cloud platform.

The POMAC system [90] by Hassan *et al.* is focused on the question, whether to offload computation to the cloud or not. Further, the approach aims at a transparent offloading process without any necessary source code changes. Thus, they present a dynamic decision engine that works at method level. Similar approaches, such as Phone2Cloud [196] and [70] by Flores *et al.*, are also focused on the offloading decision

The Avatar framework [25] is rather focused on a cloud architecture for offloading then on making decisions. As a solution, Borcea *et al.* generate a so called Avatar in the cloud for each mobile device. An Avatar is a virtualized and closely coupled representative of the local device with the same operating system. The framework provides a high level programming model in combination with a middleware. In their vision, the cloud architecture is redesigned to serve billions of mobile devices. The Avatar prototype is implemented on Android and the cloud side on an x86 Android operating system version.

4.2.4. Edge Computing Approaches

This section refers to all approaches that employ resources at the edge, including, edge, fog, and mobile edge computing. The distinction between these areas is blurry and the classification of approaches is not unambiguous. The MEC paradigm is only distantly related to the thesis, since it implicates equipping mobile network base stations with dedicated computing hardware. This allows mobile devices to offload computation to these resources within one hop (e.g., [2, 104, 133, 181]). The European Telecommunications Standards Institute published in [95] the idea of integrating MEC within the 5G network. Based on the fact that MEC requires the extension of the mobile network base stations with computing infrastructure, it is not further elaborated.

Next, the literature from three edge-related research areas is discussed: fog computing, mobile device clouds, and hybrid approaches.

Fog Computing

The major drawback of cloud computing is the latency that emerges for each communication between the local device and the cloud resource. As a solution, Satyanarayanan *et al.* proposed the concept of Cloudlets [158, 159, 160] in 2009. The idea is to reduce the communication latency by moving cloud resources to the edge of the network. Cloudlets are geographically distributed computing and storage resources that serve nearby end-user devices to augment their performance and memory capabilities. Cloudlets exploit locality and the user can reach it within the first network hop. They are deployed at public places or directly linked to base stations of the mobile network. Other researchers adopted the idea of Cloudlets and apply them in further areas, such as military [126], authentication [26, 175], and virtual reality [22]. The Cloudlet paradigm, however, requires large effort and investment into new hardware infrastructure.

In [89], Hasan *et al.* introduce Aura, an IoT-based cloud infrastructure. Aura creates a local device cloud based on IoT devices, which can be used by mobile phones in proximity to offload computation. It further facilitates the migration of data and computation transparently, in case the mobile device changes its location. As an incentive, the sharing of resource is paid with micro payments so that users devote their unused computing cycles to the system. The prototype is based on the Contiki IoT operating system and a lightweight MapReduce implementation. The system, however, only uses dedicated IoT devices for offloading and does not consider desktop PCs.

Datta *et al.* present an IoT use case for fog computing in the area of smart traffic [50, 51]. Their approach uses so called machine-to-machine gateways to connect smart vehicles with road side units or fog nodes. Based on that, the system provides the vehicles with services, such as mobility support, traffic information, and public safety announcements. The architecture has three levels: the vehicles, the road side units with fog infrastructure, and a central cloud. The vehicles provide data to the system and get access to services, depending on their location. Compared to the previous approach, Datta *et al.* present a more flexible approach for fog computing in a smart vehicle scenario. The approach, however, focuses on this particular use case and does not provide ways to integrate generic applications.

The CaRDIN [122] approach by Le *et al.* uses a combination of ARM processors and field programmable gate array (FPGA) at the edge. These FPGAs can be reconfigured to serve a certain task. CaRDIN aims at sensor-based IoT application and offers a middleware and a toolset for the integration of their FPGA hardware architecture. The system, however, is limited to the coupling of the CaRDIN hardware and software and does not support any other platform.

PiCasso [125] by Lertsinsrubtavee *et al.* offers a lightweight edge computing platform with a QoS-sensitive deployment of service at the edge. Therefore, dedicated edge nodes are deployed which host the services for other devices and monitor different contexts. So called service orchestrators accumulate the context information gathered by the nodes and decide on an optimal service allocation. The approach does not consider heterogeneous edge devices and assumes dedicated devices deployed as edge nodes.

Mobile Device Clouds

The first approach that created so called mobile device clouds (MDC) was introduced by Mtibaa *et al.* in [142]. They coordinate the collaboration of intermittently connected mobile devices to share computational capabilities. In the evaluation, they improve the responsiveness and energy consumption compared to Cloudlets and traditional cloud offloading. In [141], the authors extend the approach with power balancing across the participating mobile devices. Based on that, similar approaches arose:

The Serendipity approach [169] by Shi *et al.* provides offloading among intermittently connected devices. It differentiates between initiators and workers. The tasks are represented as blocks in a directed acyclic graph and are organized based on pre- and post-process dependencies. The scheduler uses that task knowledge and considers the churn rate of nearby devices. In their evaluation, Shi *et al.* developed an emulation environment as well as a prototype as a proof-of-concept. Serendipity, however, assumes a structural task knowledge and relies on a complex job profiling method.

FemtoClouds [86] by Habak *et al.* presents an approach for the collaboration of co-located mobile devices. The architecture is dynamic, self-configuring, and considers the churn of mobile devices. FemtoClouds assembles a MDC and focuses on the scheduling of tasks. Hence, an optimization problem is formulated that is approximated by a greedy heuristic. Similar to Serendipity, FemtoClouds are evaluated based on a simulation and a proof-of-concept prototype development.

In [139], Marinelli proposes Hyrax, an Android application that forms clusters of mobile devices. On that cluster, computationally intensive problems can be computed based on Hadoop[2], a MapReduce implementation. Hyrax uses a central server to coordinate the mobile workers, which communicate directly via 802.11g.

All MDC approaches, however, are limited to offload computation from the local device to other mobile devices in proximity. They do not consider edge devices, such as PCs equipped with GPUs, to increase the performance as well as the resource elasticity.

Hybrid Approaches

Several approaches use a so called three tier architecture [128] that combines remote cloud resources with fog and edge resources. Consequently, mobile devices at the edge, fog infrastructure in proximity of the end-user, as well as the cloud are used as computational resources in one system. Each of these three resource types has advantages and disadvantages.

In [208], Zhang *et al.* present a hybrid offloading platform that uses MDCs as well as cloud infrastructure. They aim at a higher scalability based on the elastic use of the two resource types. In addition, they reduce the energy consumption by deciding between a local execution, an ad-hoc execution on nearby devices, and an execution in the cloud. The approach, however, does not consider the execution on standard edge hardware, such as standard CPUs or GPUs.

[2]http://hadoop.apache.org

In [91], Hassan *et al.* introduce a more generic approach for fog computing. They identify the potential of edge resources to offload data as well as computation. The approach is rather focused on the decision itself, than on a concrete fog architecture. Hassan *et al.* present a variety of fog and cloud computing resources that vary in network bandwidth, latency, CPU performance, and memory capacity. Based on these characteristics, the approach makes a computation offloading decision to the most suitable resource.

CloudAware [145, 146] by Orsini *et al.* proposes a context-adaptive middleware for a mobile edge and cloud applications. It combines nearby edge and cloud infrastructure to offload computationally intensive application parts. Further, CloudAware supports elasticity and scalability for mobile applications. The execution strategy considers cloud resources, cloudlets, and a local execution as fallback. The Android-based prototype is evaluated based on the Nokia MDC data set. However, CloudAware does not consider offloading computation to other end-user devices at the edge.

In [20], Bhattcharya and De employ two sorts of resources for computation offloading. First, they use standard cloud resources and, second, end-user devices from the edge of the network. These edge devices are smartphones, tablets, routers, and laptop. Bhattcharya and De formulate a mathematical model based on directed acyclic graphs to represent the offloading problem. They compare the overall performance of the used resources based on different applications. Due to the fact that the approach is not implemented, major issues in relation to heterogeneity, network, and device fluctuation are not addressed in this approach.

To complement cloud resources, Fernando *et al.* propose a work stealing model for mobile resource clouds, called Honeybee [68]. Mobile devices form so called mobile crowds to provide a low latency computation service. Fernando identifies heterogeneity, unknown resource capabilities, and dynamism as main challenges of mobile crowds. As a solution, the Honeybee work stealing approach is applied, balances the load, and compensates the missing knowledge about processing capabilities. To cope with dynamism, Fernando *et al.* integrate fault-tolerance mechanisms that accommodate device leaves and failures. The approach does not directly consider the irregularity of jobs. Further, edge devices other than smartphones are not integrated, which leads to a large loss of computational capabilities.

4.2.5. Computation Offloading Systems

In this section, computation offloading approaches are presented. The approaches from the previous areas were focused on the infrastructure and resource perspective of a distributed computing system. In computation offloading, the focus is on the offloading decision as well as on the mechanism that facilitates it. Offloading is based on the RPC paradigm, which was first implemented by Birrell and Nelson [21] in 1984.

The MAUI approach [48] by Cuervo *et al.* presents a system for fine-grained offloading from mobile devices to a fixed infrastructure. The approach is energy-aware and minimizes the effort of application programmers. MAUI has a decision engine that decides during runtime which method should be executed remotely. In order to do that, two versions of the application exist: one on the mobile device and one running on the remote infrastructure. These two parts are connected via programming reflection and type safety to identify and transfer remote parts. During runtime, a profiler gains knowledge about the methods behavior. Using this information, the amount of data to be transferred, and the current network conditions, the decision engine chooses the local or remote execution. MAUI aims at maximizing the energy saving and reduces the method runtime. Due to the continuous profiling, MAUI is highly dynamic. However, the offloading is limited to the fixed infrastructure and the scalability of the infrastructure is not considered. Several approach are similar to MAUI and differ in certain aspects: exploitation of resource locality (e.g., [98]), more sophisticated APIs (e.g., [119]), fault-tolerance measures (e.g., [18, 189]), or code partitioning scheme (e.g., [209]).

Similar to MAUI, Chun *et al.* introduced CloneCloud [46], a mobile code offloading system. CloneCloud automatically transforms unmodified mobile applications to virtualized versions, so that they can benefit from offloading to the cloud. Therefore, it uses application-level virtualization on the mobile device as well as on the cloud resource. In particular, they use the Dalvik VM and the Java VM respectively. The system profiles the application and dynamically decides when to offload an application thread from the mobile device to the cloud. Compared to MAUI, CloneCloud supports remote execution of native functions based on the virtualization technique. It further is more transparent for the programmer and includes mechanisms for migration and merging of methods.

Both approaches, however, have shortcomings that are addressed by ThinkAir [112]. ThinkAir considers the dynamic behavior of mobile environments in a more detailed way compared to MAUI. It also focuses on a more sophisticated resource provision in the cloud to realize a higher efficiency. In comparison with CloneCloud, ThinkAir provides parallel execution on cloud resources to further increase the performance. In [14], Barbera *et al.* presented CDroid, an extension of ThinkAir, that further deploys cloud services. CDroid closely couples the cloud instance of a mobile device and synchronizes the two constantly while considering energy consumption. In case of a computationally intensive task execution, the required data is already stored on the cloud instance, which reduces the execution delay and battery consumption. A similar approach named Cuckoo [107] is presented by Kemp *et al.* Cuckoo has a looser coupling between the cloud and the mobile device, thus, it rather offloads only computationally intensive parts then having an entire application clone in the cloud. Cuckoo is Android-based and focuses on an easy-to-use application integration. However, these approaches only consider cloud or fixed infrastructure as remote resources. Further, they use closely coupled virtual images in the cloud, which makes the approaches inflexible in terms of resource migration.

The Clone2Clone approach [113] by Kosta *et al.* places device-clones on cloud resources. Based on that, not only computation is offloaded to the cloud, but also the smartphone-to-smartphone communication. Devices that cooperate with each other do not communicate directly, but rather let their device-clones in the cloud handle all communication. The unreliable and unstable ad-hoc communication between the real world devices is therefore avoided and offloaded to the cloud, where communication takes place within a high-bandwidth and low latency environment. Kosta *et al.* argue that this mechanism increases the performance and the battery life time.

The COMET approach [80] by Gorden *et al.* is an offloading approach that migrates jobs to a static computing infrastructure. It is based on the Android operating system and the Dalvik VM. Instead of using RPC to initiate the remote executions, COMET uses distributed shared memory to connect the resource with the offloading device. Thus, the approach develops a specialized version of distributed shared memory that reduces the communication between the client and server. It supports multi-threaded computation offloading and allows the migration of threads between local device and remote resource at any time. In case the connection to the remote resource is cancelled, the computation can seamlessly continue on the local device.

The approaches so far discussed make an offloading decision based on predefined policies, application parameters, and current system context. The MALMOS approach [65] by Eom *et al.* introduces a further improvement in terms of offloading decisions. It uses previous decisions in combination with their correctness. Based on that data, MALMOS applies three online machine learning algorithms, namely, instance-based learning, perceptron, and naive Bayes. It facilitates a high adaptability to network conditions as well as the computing capabilities of the participating devices. MALMOS increases the accuracy of the offloading decision by 10.9%-40.5%.

Next, workload partitioning in code offloading systems is discussed.

Workload Partitioning

In an offloading scenario, partitioning can be done in three different ways: application, code, and data partitioning. Partitioning of applications describes the deployment of distinct application parts on different devices in the environment. The partitioning of code implies splitting up tasks into parts that are offloaded and parts that are executed locally. Lastly, data partitioning means the partitioning of a task's data in several parts and running these tasks on distinct devices. These parts are not necessarily sized equally, depending on the approach. For this thesis, splitting up applications is out of scope and is therefore not covered.

The automatic code partitioning is addressed by several researchers from different domains, such as, [130] and [209]. In [110], Kopfler *et al.* present an automatic approach for code partitioning in heterogeneous environments consisting of CPUs and GPUs. They use a machine learning approach that is based on an Artificial Neural Network to predict the benefits of partitioning. This prediction model uses static program features and dynamic, input-sensitive features. As an example for a feature, the task complexity is computed during compile time or approximated, if runtime information is necessary. Internally, the approach computes all partitioning options with a 10% granularity and decides on the most promising one. The approach does not specify any type of computing environment characteristics, but is rather focused on the task perspective of workload partitioning.

The approach [182] by Travedi *et al.* is focused on data partitioning on volunteer-based grid environments. They aim at a high throughput and consider device heterogeneity in terms of performance and connectivity. The approach decomposes the workload in fine-granular and uneven parts depending on the performance of the participating devices. Faster machines receive a larger part of the workload and vice versa. The approach however, does not consider irregularity in task structures, which can also lead to bottlenecks.

4.3. Summary

This section summarizes the related work. Table 4.1 on page 44 shows the classification of the evaluated approaches from the literature. The table does not contain all examined approaches, but rather a selection of the most important representatives from each research area. First, the general observations regarding the classification are presented, which corresponds to the vertical interpretation of the table. After that, the research areas are discussed in detail, i.e., the horizontal perspective on the evaluation table.

The classification is developed referring to the derived requirements from Chapter 3 and, thus, reflects the relation between the researched literature and the thesis. Independent from the research area, heterogeneity is a major challenge and not covered by most approaches, except from accessibility heterogeneity. The reason for that fact is, that one of the most distinctive characteristic of distributed systems are the use of different network technologies. This leads to diverse bandwidths, latencies, and stability of the participating nodes. Without coping mechanisms, systems were not able to function properly. Several systems handle task heterogeneity to a limited extend, i.e., they use QoS measures to tailor computation. However, they do not consider the irregularity of tasks, which can lead to bottlenecks. The hardware and operating system heterogeneity is rarely covered by the evaluated approaches. The most intricate heterogeneity refers to the programming language.

		Heterogeneity					Edge Support		Elasticity		Usability		
	System	**Operating System**	**Hardware**	**Language**	**Accessibility**	**Task**	**Churn**	**Mobility**	**Adaptability**	**Efficiency**	**Abstraction**	**Transparency**	**Obtrusiveness**
Cluster	MapReduce [52, 53]		○		○				○	●	●	●	
	Spark [205, 206]	○	○		○	○			○	●	●	○	
	Dryad [99]				○	○			○	●	●		
	LibWater [81]		●			○				●	●	●	
Grid	OurGrid [7]				○						●	●	
	SpeQuloS [54, 55]				○	○			●	○	○	●	○
	GridBot [172]	○	●		●				○	●	○	●	
Volunteer	HTCondor [132, 179]				●	○	○			○		○	○
	Spawn [187]				●	○	○		○	○		○	
	BOINC [5, 6]		●		●		○			○		○	○
	XtremWeb [66, 37]		●		●		●			○		○	
	Entropia [43, 35]				●	○	○					●	●
	Nebula [40, 157, 103]	○			●	○	○		●	●		○	
	Ren *et al.* [152]						●		○				
Cloud	Justino *et al.* [105]				●				●	●	○	●	
	POMAC [90]				●	○			●			●	
	Borcea *et al.* [25]				●	○			●		●	●	
Edge	Cloudlets [158, 159, 160]	○			○	○		○			○	●	
	Aura [89]		○	○	○		●	○	○		○	●	○
	Datta *et al.* [50, 51]							○	●			○	
	PiCasso [125]						○		○			○	
	Mtibaa *et al.* [141, 142]				●	○	●	●	○			●	○
	Serendipity [169]				●	○	○	●	○		○	●	
	FemtoClouds [86]				●	○	●	○	○			○	
	Hyrax [139]				●		○	○		○	○	●	
	Zhang *et al.* [208]				●	○	○	○	●	○	○	●	○
	Hassan *et al.* [91]				○		○	○	○			○	
	CloudAware [145, 146]	○	○		○		○	●	○	○		●	
	Bhattcharya [20] *et al.*	○	○		○	○	○	○	○				
	Fernando *et al.* [68]	○	●		○	○	●	●	○		○	●	
Offloading	MAUI [48]				○							●	
	CloneCloud [46]										●	●	
	ThinkAir [112]								○	●	●	●	
	Clone2Clone [113]				○				○	●	●	●	
	COMET [80]		○	○							●	●	
	MALMOS [65]					○			○	○	○	●	
	Kofler *et al.* [110]	○	●			○			●	●	●	●	
	Trivedi *et al.* [182]	●	○		●	○			○				

Table 4.1.: Classification of related work. The result of the literature review introduces the research gap for the thesis. None of the presented approaches fulfills the requirements. The table encompasses the most important approaches as representatives for each research area. (● - fulfilled, ○ - partially fulfilled or mentioned without further specification.)

The support of edge resources is only covered by the respective fields of volunteer and edge computing. Both largely cover the churn of devices. Their mobility is primarily covered by edge approaches. The resource elasticity in terms of adaptability and efficiency are handled by diverse approaches from different fields.

Regarding usability, many of the evaluated approaches introduce a programming abstraction as well as distribution transparency. However, the interference with the resource providers is not considered by the most frameworks. Next, the research areas are individually discussed in more detail.

In general, cluster computing approaches are based on homogeneous computing infrastructure that is closely coupled with high capacity networks. As a result, these systems do not require mechanisms to cope with heterogeneity, except from performance. These performance gaps are classified as hardware heterogeneity and cluster computing approaches often apply work stealing mechanism, to ensure weighted workload balancing. As depicted in Table 4.1, the focus of cluster computing is on efficiency and computation abstraction to facilitate high performance computing.

Grid computing environments consist of distributed and dedicated computing resources, which are comparable to clusters in terms of heterogeneity. Especially accessibility heterogeneity as well as transparency are tackled by these research approaches. The infrastructure is static and less efficient in terms of elasticity. Further, grid computing approaches do not support generic applications, but rather single-application environments.

In volunteer computing, user-controlled edge devices are used as computing resources. These approaches especially cope with accessibility and churn of devices. The level of usability of volunteer computing approaches is low and often only single-application systems are supported with no abstraction for computation. Since the resources in these systems are stationary, these approaches do not support mechanisms to cope with device mobility.

In cloud computing, a fixed and homogeneous computing infrastructure is used. Therefore, these approaches are rather focused on the resource consumers than on the providers side. Especially, the transparencies as well as the resource elasticity are covered in great detail.

Edge approaches cope with device churn as well as mobility, but they do not cover heterogeneity issues. The resource elasticity is partially covered by the majority of the approaches in contrast to the efficiency. In terms of usability, edge computing approaches rarely provide computation abstraction or solutions for unobtrusiveness. Similar to cloud computing, fog computing relies on dedicated infrastructure that is deployed in proximity of the users. On the other side, MDCs solely use smartphones of end-users as resource providers. Also the hybrid approaches mostly combine fog and MDC resources and do not consider the full range of user-controlled edge devices. Many approaches in this area are specific for a certain use case and do not offer frameworks for the integration of generic applications.

Offloading approaches provide computation abstractions, transparencies, and offloading decision support. Mostly, static infrastructure or cloud resources are used as providers and no edge devices are incorporated. Similar to cloud computing, the resource perspective on these approaches is not given.

As visualized by Table 4.1, none of the presented approaches from the examined research areas fulfills the requirements, leading to the research gap which is covered by this thesis. Especially overcoming different heterogeneities and the support of edge devices are two major issues, which are rarely addressed together by the same approach. Therefore the objective of the thesis '*the development of a computation placement framework that provides an abstraction for computation and resource elasticity in edge-based environments.*' is not achieved by approaches from literature.

This chapter discussed the related work of the thesis. Therefore, a classification was developed based on the research requirements from Chapter 3. The next chapter introduces a system design to close the identified research gap by answering the research question and meeting the requirements.

5. Elastic Computation Placement in Edge-based Environments

After the literature analysis which identified the research gap, the following chapter presents the design to answer the stated research questions and to fulfill the requirements from Chapter 3. The design of the thesis is twofold. The first part is the Tasklet system that is a computation placement framework. It is the foundation for construction, execution, and distribution of independent computational units – so called Tasklets. The Tasklet system is designed in compliance with the requirements of distributed systems, however, it is not designed to support edge environments in particular. The second part of the design is an edge support layer that extends the Tasklet system to handle the characteristics of edge devices.

The remainder of this chapter is structured as follows: First, the design principles as well as the overall architecture are presented. Second, the application model is introduced which categorizes applications and determines the general assumptions for computation placement. Third, the Tasklet system and fourth, the edge support layer are introduced.

5.1. System Overview

The system overview consists of two parts: First, the general design principles and the system design are shown. Second, the overall system architecture is introduced.

5.1.1. Design

Multiple computation offloading approaches exist which could potentially be extended to solve the stated research questions. However, most of these approaches are problem specific and work only with a certain type of resource (e.g., [48, 112,

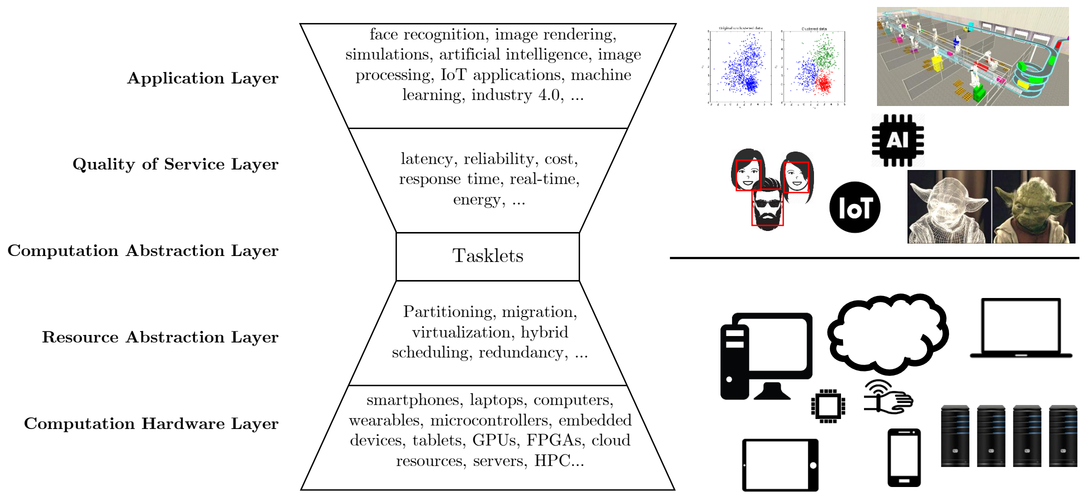

Application Layer
Quality of Service Layer
Computation Abstraction Layer
Resource Abstraction Layer
Computation Hardware Layer
face recognition, image rendering, simulations, artificial intelligence, image processing, IoT applications, machine learning, industry 4.0, ...
latency, reliability, cost, response time, real-time, energy, ...
Tasklets
Partitioning, migration, virtualization, hybrid scheduling, redundancy, ...
smartphones, laptops, computers, wearables, microcontrollers, embedded devices, tablets, GPUs, FPGAs, cloud resources, servers, HPC...
AI
IoT

Figure 5.1.: System concept based on the hourglass model of the internet architecture. The overall purpose is to offload computationally intensive parts of generic applications to heterogeneous edge computing hardware (right side). The concept consists of five layers, each of which provides another level of service abstraction. In the center of the model, Tasklets create the best-effort based, lightweight, and interoperable computation abstraction, which is surrounded by the resource abstraction and the quality of service layer, that integrate edge resource and execution guarantees, respectively.

113]), a fixed infrastructures (e.g., [3, 53, 111]), or for specific applications (e.g., [6, 53, 200]). Furthermore, most of the approaches are not built to be extended with algorithms and mechanisms to cope with the stated challenges: They do not offer a suitable level of flexibility for applications and adaptability for altering contexts. Consequently, the presented design cannot extend an existing system. Therefore, it is a fully-fledged system approach, consisting of the Tasklet system and the edge support layer.

Certain design principles can be derived from the requirements. The overall goal is to facilitate task offloading from generic applications to heterogeneous edge computing resources. Figure 5.1 shows the main concept of the approach. The design is inspired by the Internet architecture, where the IP is the link between various lower layer technologies and upper level application protocols. On both ends of this architecture, the variety of applications and protocols is huge. In the center, however, the IP facilitates the core functionalities and establishes interoperability. The IP is lightweight and offers best-effort service, which means that it does not provide any communication guarantees. Higher layer protocols have this responsibility. In case of the Internet architecture, physical layer and data link protocols are on the bottom of the model. These protocols facilitate basic one-to-one communication. In the present computation placement architecture, the lowest layer consists of computation hardware such as PCs, GPUs, or smartphones. The resource abstraction layer's responsibilities are to abstract the functionality of the lower layer and to overcome edge resource specific issues like heterogeneity, faults, and fluctuation. Therefore, hardware virtualization and algorithms for partitioning, migration, context-awareness, fault-tolerance, and hybrid scheduling are enforced on this layer. The computation abstraction layer is located in the center of the model. Analogous to the IP, it is lightweight, provides interoperability, and is extensible. This layer consists of Tasklets, which offer best-effort computation placement. On top of Tasklets, a quality of service layer adds application tailored guarantees, which optimize the task allocation. Finally, the application layer interacts with all kinds of applications that are suitable for offloading. The overall approach is created from scratch and offers full flexibility and adaptability on every level. The main design principles of the system are further elaborated:

Lightweight: The major focus of this thesis are edge environments which consist of heterogeneous devices. For their integration of the thin devices, a lightweight runtime environment and architecture are crucial.

Interoperable: The aim is to integrate all kinds of resources into one computation placement system. Therefore, tasks are allocated to various resources, which requires the shipment and remote execution of tasks. Interoperability between these resources facilitates this task exchange.

Portable: The device heterogeneity entails various computing platforms and architectures like GPUs, CPUs, or System on a Chip. All these physical devices operate differently in terms of memory, instructions, and parallelism. To cope with this diversity, a common abstraction for computation must be established.

Extensible: In contrast to a standard task offloading system, which assumes grid or cluster resources, the present system needs to be extensible in terms of algorithms and mechanisms that enable the utilization of edge resources. The specific characteristics of these resources require measures that are added to the general functionality of the system componentwise.

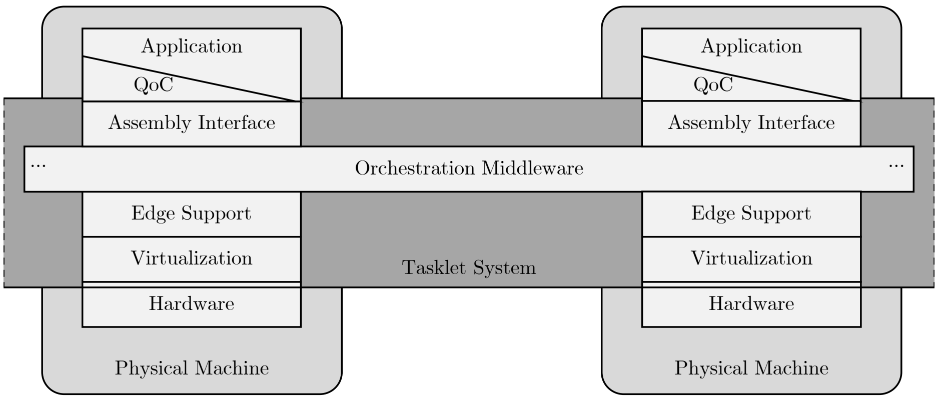


Figure 5.2.: Overall system architecture. It consists of four layers, which sit on top of the computation hardware and below the consumer application. It is composed of an assembly interface, an orchestration middleware, an edge support, and a virtualization layer.

5.1.2. System Architecture

Figure 5.2 shows the overall system architecture. Each physical machine runs the Tasklet middleware. It connects all participating devices in the system. On top of the layered architecture, the consumer application initiates computationally intensive tasks. Depending on the required quality level, the application uses the so called Quality of Computation (QoC) layer to tailor the quality of service level of remote task executions. Based on that, an assembly interface connects the application with the rest of the Tasklet system. The orchestration middleware allocates tasks in the system and enforces the stated QoC goals. The hardware layer supports various types like CPUs, GPUs, smartphones, and microcontrollers. To abstract the computational capabilities of these physical machines, a virtualization layer handles hardware heterogeneity and offers homogeneous computation. Compared to traditional offloading resources, edge resources are more volatile, unpredictable, and erroneous. Therefore, the edge support layer offers stability, reliability, and homogeneity in terms of computational performance.

The next chapter derives an application model to classify applications in terms of their offloading capabilities.

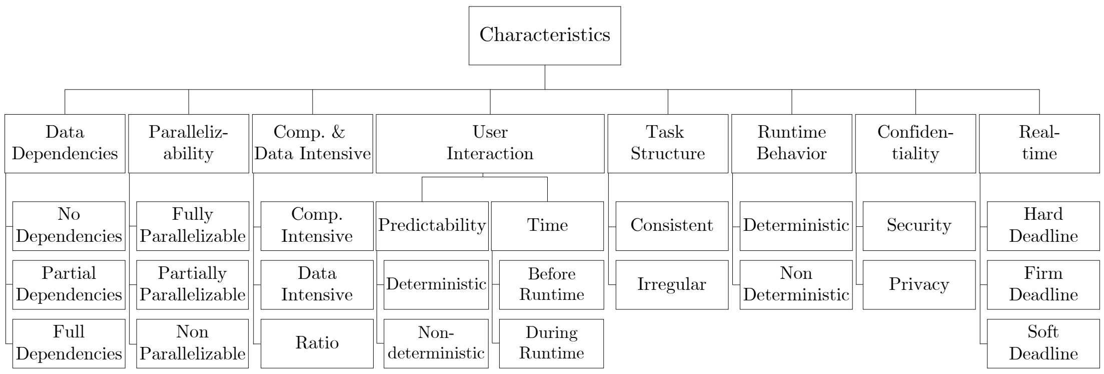


Figure 5.3.: Taxonomy of applications that are offloading candidates. The eight characteristics can have multiple dimensions.

5.2. Application Model

Since not all applications are suitable for computation placement systems, this section categorizes applications and narrows them down considering the focus of this thesis. First, the relation of applications, tasks, and subtasks is defined. After that, the taxonomy of applications for computation placement is introduced. Finally, a conclusion for the present thesis is drawn.

The computationally intensive parts of a consumer application are defined as tasks. Computation placement systems allocate these tasks on remote machines to save local resources. To enhance the benefit, some tasks can be split up into several subtasks. This increases the parallel computation and reduces the response times. All subtasks are required to assemble the overall result of a task, which is then submitted to the consumer application. Therefore, the last subtask that arrives determines the task completion time.

5.2.1. Taxonomy

Figure 5.3 shows the taxonomy of applications. This taxonomy is not exhaustive, but rather describes the relevant characteristics of applications that are offloading candidates. The main characteristics are data dependency, parallelizability, computational intensity, data intensity, user interaction, task structure, runtime behavior, confidentiality, and real-time. Characteristics can have multiple dimensions that each describe a specific application behavior. Next, all characteristics are presented in detail.

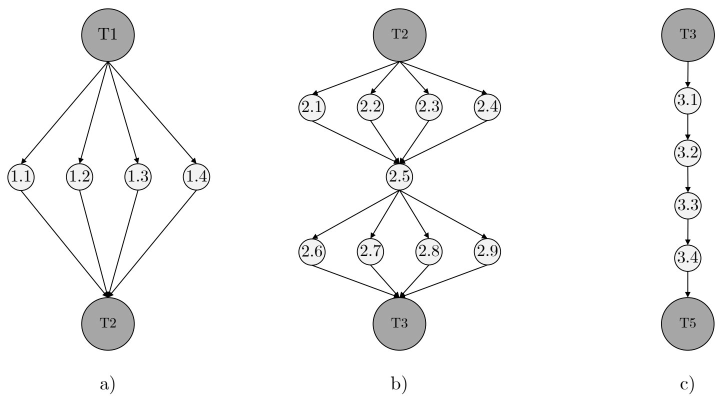


Figure 5.4.: Taxonomy of parallelizability of tasks. a) shows fully parallelizable tasks where all computationally intensive subtasks (in light gray) are executed in parallel. In b), the structure is partially parallelizable, since there are serial computations necessary. In c), the tasks are completely serial, and therefore there is no potential performance gain due to parallel executions.

Data dependencies characterize the execution constraints of tasks on data level. The three dimensions are: (1) *no dependencies*, (2) *full dependencies*, and (3) *partial dependencies*. In case the data of a task consists of 100 chunks and each chunk can be processed in an individual subtask without accessing other parts of the data, the data has no dependencies. An example are image rendering applications like ray tracing, where each pixel can be computed individually. This kind of application benefits from offloading. For tasks that have full data dependencies, each subtask needs the entire data for the computation of the subproblem. An example for that is a face recognition approach, where each subtask compares an image against the database. In this case, all subtasks need access to the complete data, which reduces the benefit of computation placement drastically. Application data with partial dependencies are in between these two cases, meaning, that for each subtask execution, a subset of the data is necessary. This is the case for image filtering algorithms, especially for those, where for the computation of a pixel each adjacent pixel is necessary.

Parallelizability describes how well the workflow of a task can be parallelized. Compared to the data dependencies, the parallelizability refers to the process flow of a task and has no relation to data. Figure 5.4 shows the three dimensions, namely, *fully parallelizable*, *partially parallelizable*, and *non parallelizable* or serial. The first dimension are tasks that are *fully parallelizable* which has the largest potential offloading benefit. The workflow of these tasks has no serial parts, which is often the case for image rendering applications. The second dimension is *partially parallelizable*, meaning, that the workflow of the computationally intensive parts of an applications has serial parts. This is shown in Figure 5.4 b). Therefore, the execution consists of different stages. Some of them can be executed in parallel and others are serial. For that, the intermediate results are collected and the serial part is either executed locally or remotely. In case another parallel part follows, the task can be split up again and remotely executed in parallel. It depends on the ratio between parallel and serial stages of a workflow if a task is suitable for parallel task offloading. A task's workflow can also be entirely serial, as shown in Figure 5.4 c). With no parallel execution, computation placement is conditionally beneficial. For example, if long-running serial executions are computed on a very powerful resource, the time and local energy consumption can be improved.

Computational and data intensity are two relevant characteristics. Especially their relation determines if applications can benefit from remote placement. Tasks that are computationally intensive with only a small amount of data are suitable, since the transfer costs are neglectable. However, the placement of tasks that require a large amount of data for processing can be rather inefficient. As a solution for computationally and data intensive tasks, a data management system can distribute the data before the task execution. By applying different data distribution strategies, the system copes with the overhead of data distribution.

User interaction limits the capabilities for remote task execution. Although the user interaction takes place on the resource consumer it can be transferred to the remote resource providers at overhead costs. The point in time of a user interaction can be deterministic or non-deterministic. Further, some interactions can be made by the user before runtime of a task and others only during the task runtime, since they depend on an intermediate result. Therefore, the user needs information to make a decision.

Task structure describes the computational intensity of a task throughout its parameter range. There are two different types of tasks in terms of task structure, consistent and irregular. A consistent structure implies, that over the entire parameter range, the computational effort is equal or at least similar. Applications which are based on Monte Carlo simulations where several thousand simulations are executed within a task are consistent – see Figure 5.5 a). Contrary

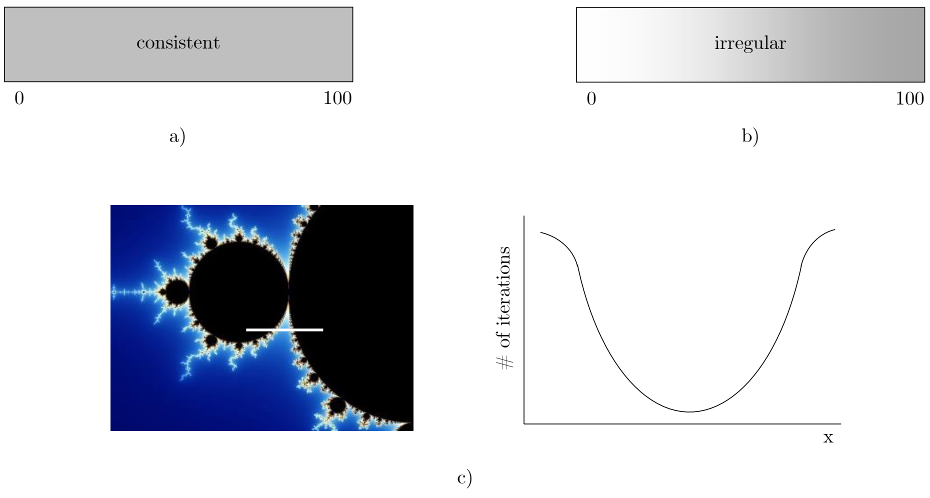


Figure 5.5.: Consistent and irregular task structures. a) shows a consistent task complexity, indicating, that each segment of a task is equal in terms of computational complexity. b) shows an irregular task complexity, meaning, that the computation of the segment $0 - 10$ is less complex compared to $90 - 100$. c) Shows an example based on the task structure of the Mandelbrot set. The left side of c) shows a part of a MBS image with a marked section. On the right side, the graph shows the computational effort that is required for the computation of this part of the MBS.

to that, an irregular task structure implies that the computational effort varies throughout the parameter range, as shown in Figure 5.5 b). An example for that is the computation of a Mandebrot set (MBS), where the color of each pixel can be calculated with complex numbers. Figure 5.5 c) shows the relation between the computed section of the MBS and the respective computational effort in form of iterations. Irregular task behavior can lead to bottlenecks when splitting up tasks in several subtasks and offloading them on remote resource providers.

Runtime behavior is another general characteristic of applications. It is distinguished in two different dimensions, deterministic and non-deterministic. A deterministic task always produces the same output, assuming the same parameters. Therefore, it has no random factor. A non-deterministic task, on the other hand, can have different results, even with the same input parameters. This limits the predictability of the task's runtime.

Confidentiality is a term for data security and privacy. Some applications require a certain level of security and privacy when dealing with sensitive data. Many approaches take care of these issues in distributed computing systems and especially for offloading systems.

Real-time task execution is required by some applications. Real-time in general means that a task execution is timely and has to be done within a certain time interval to retain the functionality of the application [29, p. 2 f]. There are three different kinds of real-time: First, hard real time, which implies that no deadline can be missed without causing a total system failure. Hard real-time systems are used in industrial process controller or medical devices, like heart pacemakers. Second, firm real-time deadlines, meaning, that missing a few deadlines is tolerable but degrades the quality of the system. Further, results that miss the deadline cannot be used. The third type of real-time assumes soft deadlines, implying that the application quality decreases after a deadline is missed, but the results are still used. Real-time is hard to guarantee in offloading scenarios.

5.2.2. Application Model Summary

The application model summarizes all assumptions regarding the offloading suitability of applications. Considering the data dependencies, the presented approach is not limited, however, the benefit increases with decreasing data dependencies. Thus, the approach does not introduce algorithms to reduce the impact of data dependencies. This is analogous to the parallelizability of tasks. The placement mechanism of the basic Tasklet system can also be used for serial task offloading, which is not the focus of this thesis. This area is well researched by systems like MAUI [48] or Thinkair [112], and focuses on offloading strategies that consider energy consumption. Fully parallel tasks exploit the entire benefit.

Task offloading is beneficial for computationally intensive applications. Data intensive application, however, require an appropriate data management which is not in the focus of this thesis. The approach assumes that user interactions are made before runtime and interaction during runtime are excluded from the design. One main focus of this thesis is to overcome heterogeneity. Therefore, irregularity of task structures is considered by the present approach. In terms of the runtime behavior, the approach encompasses deterministic and non-deterministic applications. Security and privacy are not in focus of this thesis as well as hard and firm real-time task execution. However, for soft real-time applications, the approach is applicable.

5.3. The Tasklet System

This section is based on [166][1] and introduces the Tasklet system, which is the foundation of the thesis' design. The section is structured as follows: After a brief characterization of Tasklets, it describes the domain of Tasklet applications. Next, the section introduces the Tasklet system model, the Tasklet lifecycle, and the Tasklet middleware in depth. Thereto, the three layers of the Tasklet system are explained, namely, the construction, execution, and distribution layer. Finally, the Quality of Computation paradigm is briefly introduced, which is an execution quality concept for Tasklets.

5.3.1. Tasklets

Tasklets are *fine-grained units of computation* that make use of heterogeneous processing entities. They can run on many idle computational resource, regardless of their architecture, operating system, location, or network connection. Thus, Tasklets overcome the heterogeneity of computing environments to aggregate computational capabilities.

Tasklets are not entire programs, but *extracted subroutines* of computationally intensive applications. The duration of Tasklet executions is not fixed and can range from few seconds to several minutes. To use Tasklets, programmers identify computationally intensive parts of applications and transfer them into Tasklets. The Tasklet API and an Eclipse Plug-in support this process. During runtime, these applications initiate a Tasklet request and hand over the control of execution to the Tasklet middleware.

By default, the Tasklet system offers *best-effort computation.* It is neither evident for the developer where and how the execution takes place, nor whether the Tasklet is executed at all. This defines the lowest possible degree of execution quality for a Tasklet. While this is sufficient for some applications, most applications need further guarantees for the execution of Tasklets. To tailor the computation to these requirements, developers can set so called *Quality of Computation (QoC)* goals for each Tasklet. The Tasklet system supports different *QoC goals*, such as *reliability*, *cost*, *privacy*, *speed*, and *multiple execution.* QoC goals are enforced in

[1][166] is joint work with J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker

the Tasklet middleware. This is done transparently to the user and the application programmer. Depending on the current environmental context and on the QoC goals that are set by the developer, the middleware decides on the most promising execution strategy. The environmental context contains the amount, stability, performance, mobility, and distance of available devices.

Tasklets are *self-contained*, meaning, that they include source code, data, parameters, and QoC information. Thus, a Tasklet is transferred with all components that are necessary for the execution. With the Tasklet system, computationally intensive parts of an application can be further split up into several Tasklets and executed in parallel. This can be done automatically, semi-automatically, and manually, depending on the application's structure. The Tasklet middleware distributes the subtasks and returns a single result in the end.

To define the logic of a Tasklet, the application programmer writes the source code in *C--*, the Tasklet language. C-- is a C-like procedural programming language, which is especially designed for remote executions. It is designed from scratch to consider special requirements and allow adaptability and extensibility. The middleware assembles Tasklets as closures. Thus, they contain all necessary elements for execution. This also includes data for the computation, which is sent along with each Tasklet. The Tasklet system decides, where to execute a Tasklet and possibly ships it to remote resource providers. After the Tasklet arrived, the so called *Tasklet Virtual Machine* is responsible for its execution.

5.3.2. Tasklet System Model

The Tasklet system model consists of three types of entities: *resource consumers*, *resource providers*, and *resource brokers.* Resource consumers run computationally intensive applications and initiate Tasklet executions to offload complex application parts to remote machines. These machines are called resource providers and they contribute their computational capabilities to the system in form of *Tasklet Virtual Machines (TVM).* Each provider can start multiple instances of the TVM to allow concurrent Tasklet executions. A participating node can be both, a provider and a consumer, at the same time. Resource brokers are responsible for the resource management in the Tasklet system and run on dedicated trusted server. Depending on the number of concurrent providers and consumers, the system

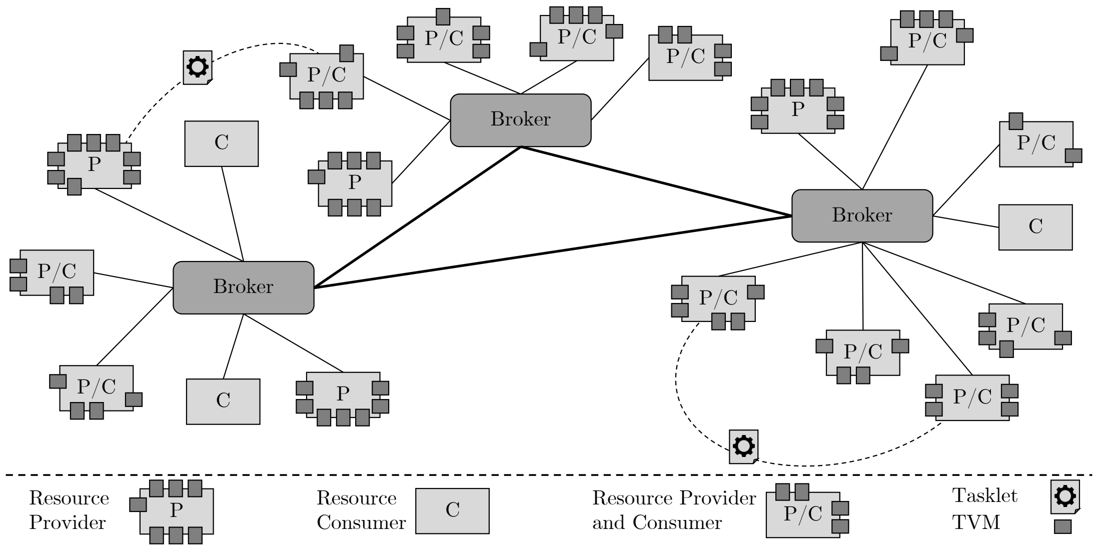


Figure 5.6.: The Tasklet system model. Resource providers (P) offer their computation power in the form of Tasklet Virtual Machines (TVMs) to resource consumers (C). The peer-to-peer broker overlay performs the scheduling. Providers and consumers exchange Tasklets and results directly with each other.

scales the number of brokers and balances the resource management workload among them. The brokers communicate via a broker-overlay to exchange state information. To join the system, providers register at a broker with benchmark information and the number of virtual machines they contribute. Therefore, providers as well as consumers need the address of at least one broker for the bootstrapping. For the initiation of a Tasklet execution, the resource consumer requests resources from the broker. This resource request contains the level of QoC that the execution requires. Based on these information and the current state of the system, the broker selects suitable resources for the tasks. The broker replies to the consumer with the respective address and performance information. After that, the exchange of Tasklets and results is done directly between the consumer and provider. In order to achieve the cooperation of the system entities each of them runs an instance of the Tasklet middleware. Figure 5.6 shows the system model of the Tasklet system.

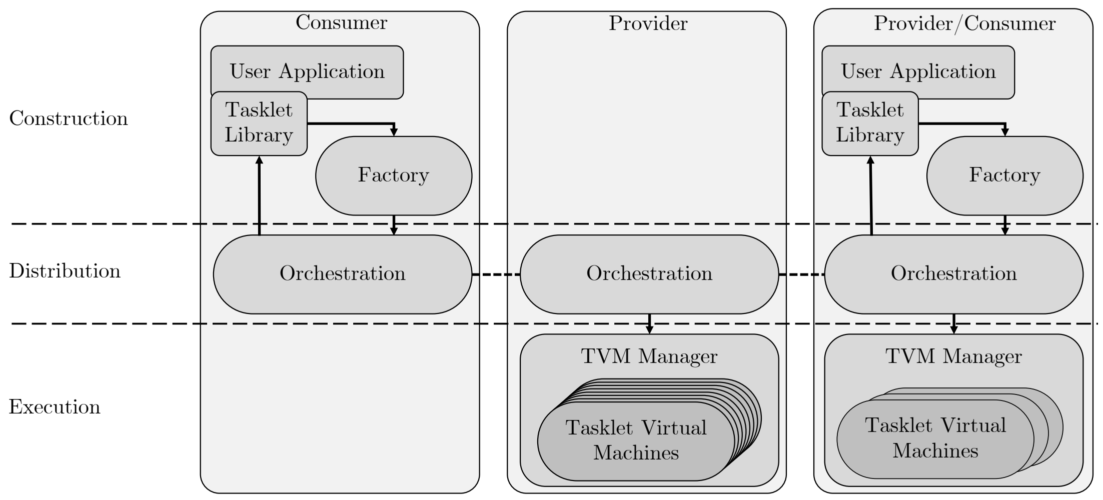


Figure 5.7.: The architecture of the Tasklet middleware. An application initiates the creation and execution of a Tasklet. The factory compiles C-- source code and assembles the Tasklet to a self-contained unit of computation. The distribution layer forwards Tasklets to resource providers and returns the results to the user application. The Tasklet execution is performed by virtual machines.

5.3.3. Tasklet Middleware

The Tasklet middleware consists of three layers: *construction* layer, *execution* layer, and *distribution* layer. Figure 5.7 pictures the structure of the middleware on consumers, providers, and on nodes that have both roles. The resource consumer runs an user application, which offloads computationally intensive parts via the Tasklet system. This application utilizes the Tasklet *library* that offers an API for creating, parameterizing, and starting Tasklets. Further it handles all communication with other components and forwards the plain and unprocessed data of a Tasklet request to the *factory.* The factory then compiles the C-- source code to bytecode, assembles the Tasklet as a closure, and delivers the Tasklet to the distribution layer. In the distribution layer, the Tasklet *orchestration* links the nodes in the system and is responsible for requesting resources from the broker and communicating Tasklets and their results. After a Tasklet is delivered to the assigned resource provider, the local orchestration forwards it to the execution layer. The local resources are managed by a *TVM manager* that assigns the Tasklets to a TVM for execution. The middleware transfers the results back to the consumers orchestration after the execution. The orchestration gathers all results of a Tasklet and forwards them via the Tasklet library to the application. In the following, each layer and its components are discussed in depth.

5.3.4. Construction Layer

The construction layer is the interface between the application and the Tasklet middleware. For the construction of Tasklets, two options exist: first, the integration into a general language and, second, using a factory approach for the construction of Tasklets. With the first option, overcoming programming language heterogeneity is not possible, since the construction functionality is done in the consumer application directly. The presented design uses the second option, due to the flexibility and interoperability.

As a result, the construction layer consists of the user application, the Tasklet library, and the Tasklet factory. Consumer applications utilize the library and use its API to initiate and forward Tasklet requests to the factory. The Tasklet library is language-specific and, therefore, needs to be implemented for each programing language individually. These languages are also called host languages. The host language concept describes the integration of the Tasklet environment into another language. Tasklets can also be sent directly to the factory via Sockets without using the Tasklet library. However, using the library is more comfortable and hides several complexities from the developer.

Next, the Tasklet factory, the Tasklet library, the Tasklet compiler, and the Tasklet language are described.

Tasklet Factory

The host application submits the Tasklet request to the middleware and the Tasklet factory assembles the Tasklet. The Tasklet request is a yet unassembled Tasklet which is sent from the consumer application to the Tasklet middleware and is, henceforth, referred to as *plain Tasklet.* After receiving the plain Tasklet the workflow of the factory involves the following steps: (1) retrieving and unmarshalling the plain Tasklet data, (2) checking the data for integrity, (3) extracting the source code, (4) running the Tasklet compiler to generate bytecode,

(5) checking for compiling errors, (6) assembling the Tasklet closure, and (7) forwarding the Tasklet to the orchestration. The Tasklet factory can serve multiple applications concurrently.

One feature of the Tasklet factory is the so called *bytecode caching.* Most applications send a sequence of the same type of Tasklet at irregular time intervals. In these cases, the parameters, the data, and the QoC goals may change, but not the source code. The middleware caches the compiled bytecode of each Tasklet. After the initial full transmission, the application sends a special message, which omits the source code transmission. The middleware retrieves the bytecode from the cache and substitutes the parameters directly in the bytecode. This eliminates not only the redundant transmission of the source code, but also the compilation process.

The Tasklet Library

The main focus of the library is to ease the use of Tasklets by reducing the integration overhead and lower the development effort. Therefore, it offers an API to create, parameterize, and start Tasklets, as well as to retrieve their results. From the application developer's perspective, the Tasklet integration takes place as follows: (1) identifying computationally intensive parts, (2) re-writing the logic in the Tasklet language, (3) including the Tasklet library, and (4) replacing the computationally intensive parts with the Tasklet API calls.

Besides, the goal is to allow any language to use Tasklets based on the *host language concept.* To exchange data between the middleware and the consumer application, inter-process communication is required. Several options are possible, for example, message queues, files system, pipes, message passing, shared memory, or sockets. Also higher level middleware-based abstractions, such as CORBA, are an alternative. However, not all of them are available on all operating systems and others are not well-integrated into programming language libraries. Sockets are widely supported by programming languages, well known by application programmers, and offer flexibility as well as communication performance. Therefore, the library is based on Sockets and the Tasklet communication protocol. The developer can write code in the most convenient language and use the Tasklet language for computationally intensive subtasks. This approach embeds the

Tasklet system into each host language that supports Sockets. For the use of the Tasklet system, the application programmer has two options: the manual mode and the library-supported mode. In the manual mode, the developer has to construct byte arrays that represent the plain Tasklet request manually. Next, the developer has to send a message over TCP that contains the Tasklet request. This message has to conform to the Tasklet protocol and has to be sent to the local middleware. The middleware returns a result handle and after the execution is finished, the results are retrieved over the TCP connection.

Writing the messages by hand can be cumbersome for developers. The library-supported mode helps developers to communicate with the Tasklet middleware while hiding complexities. The library offers an API to create and manage Tasklets, as well as handling Tasklet results. To create a Tasklet, the developer passes a source code file with the Tasklet logic via the API. Next, parameters and data can be attached to the Tasklet. Application tasks can also consist of several parts, each represented by a single Tasklet. These Tasklets are packed into a so called *Tasklet bundle* and represent the overall application task. To define the required quality of service level, the developer can set QoC goals for a Tasklet. After the Tasklet is started, the library API offers functions to handle all Tasklet results. The library-based solution is more comfortable for developers, however, the library must be developed for each language individually.

Tasklet Compiler

The Tasklet compiler generates bytecode from Tasklet language source code. It is designed from scratch to enable high flexibility and adaptability in terms of the language and bytecode formats. The compiler validates the source code while the bytecode is generated. This includes a type checking for the variables that are passed over from the host language. The bytecode format is lightweight to reduce the traffic of Tasklet transmissions. After the compilation process, the factory checks for compiling errors and sends feedback to the consumer application, if necessary. In case of an error-free compilation process, the factory starts the Tasklet closure assembly.

Tasklet Language

The main idea of Tasklets is to execute computationally intensive parts of applications on remote resource providers. In order to do this, the logic of these parts needs to be transferred into the Tasklet representation. For the design of this representation three options arise: (1) using the native language of each individual consumer application, (2) using an existing language to exchange computationally intensive parts, or (3) developing a new language from scratch. The benefits of the first approach are the native execution performance and that developers must not rewrite the Tasklet logic in a second language. However, brokers can only select providers that offer the same execution environment as the one the consumer application runs in. Moreover, for each language and runtime environment, a Tasklets integration must be implemented that enables remote executions.

The second alternative uses a single existing language, which is adapted for the use of Tasklets. The benefit of this solution is that all Tasklets can be executed on all participating providers, since they have the same local execution environment. However, several changes to the language are necessary to implement features like QoC or fault tolerance. Many programming languages are not designed for remote execution. Besides, the adaptability and extensibility are major issues. To realize the Tasklet design with a third party language, many features of a language are not viable. For example, paradigms and tools like object orientation, complex data types, and native file system libraries complicate the transfer of tasks to a remote provider. Object orientation potentially increases the overhead for task transmissions. Further, these languages offer files system access, which may leads to privacy as well as security issues and allows malicious behavior.

The third approach introduces a new programming language that is especially designed for remote execution and is implemented from scratch. Developers have to identify and extract the computationally intensive parts of applications and rewrite the code in the Tasklet language, similar to the second alternative. The benefit is that the language can be tailored to the requirements of the Tasklet system. Thus, it is fully adaptable to features like QoC, task migration, and data partitioning. One disadvantage of this solution is the overhead of rewriting the code as well as of supporting an interface between the Tasklet language and the original programming language an application is written in. Moreover,

```
int lower, upper, result;

procedure int checkprime (int a){…}

>>upper;
>>lower;

while(lower<upper){
 result := checkprime(lower);
 if(result # 0){
  <<result;
 }
 lower++;
}
```

Figure 5.8.: An example for a C-- source code. This code computes all prime numbers within the interval between *lower* and *upper*.

developing a domain specific language for task offloading implicates high design and implementation effort. For a research design, adaptability and extensibility are crucial, since adjustments are common. The first two alternatives have several constraints that may interfere with the requirements of the thesis. This especially applies to the requirement REQ_{F5} *overcoming edge heterogeneity*. Therefore, the design decision is made in favor of the third approach, which results in developing the Tasklet language C--, a lightweight language for remote task executions.

C-- is built to support distributed, generic, and lightweight computation. C-- code represents a single thread of computation in an imperative manner. Further, it provides some additional features to break the boundaries of remote execution, meaning, it is specifically designed to be compiled and executed on different physical machines. It is not standalone, but programming languages can integrate C-- as host languages. Thus, it overcomes programming language heterogeneity in distributed computing environments.

The goal of the Tasklet Language is to express frequently-computed subtasks of algorithms while hiding the complexity of remote execution from the application programmer. Hence, major design goals of the language are to separate the compilation and execution environment, to allow easy marshalling, and to manage the trade-off between a powerful and lightweight language. The structure of programs written in C-- is predefined to improve readability and reduce the potential for programming errors. Each program starts with a definition of global variables and constants. After that, arbitrarily nested procedures follow, which consist of variable and inner procedure definitions, a sequence of statements, and a return value, if required. The Tasklet code ends with a collection of statements that represent the main function, meaning, the starting point of the Tasklet. Figure 5.8 shows a simplified example source code. It neither offers memory pointers nor a system call library, since sandboxing is an important design goal for a code placement system. With the ability to execute system calls on a remote provider, malicious Tasklets are able to compromise the provider. Further, this eases the marshalling of Tasklet bytecode.

As the execution of a Tasklet is usually not local, the language has no standard I/O functions. Therefore, input parameters that are used during the task execution are integrated in the Tasklet closure. This applies only to inputs, which are already known during the compile time of a Tasklet. Other inputs depend on intermediate results and are transferred later on. The output of a Tasklet is aggregated during its execution on the provider side and returned to the consumer afterwards. To include the execution parameters and create results, the Tasklet language offers two remote I/O operators. These two operators bridge the gap between the consumer application and the remote execution provider.

The Tasklet input operator >> transfers input parameters from the host language to the Tasklet logic. Thus, variables in the C-- source code can be initialized with values from the host application. Therefore, the initiating application assembles a list with all Tasklet parameters. For that, explicit data type information is stated as well as a consistent variable naming throughout both programing languages. During the byte code generation, the compiler inserts the parameters from the consumer application's list into the byte code. After that, the byte code is ready for execution. In Figure 5.8, the variable values of *lower* and *upper* are passed on from the host language to the Tasklet source code. The byte code is also cached by the middleware for the code reuse mechanism. In order to do that, the parameters are substituted with placeholders. In case of a byte code reuse, these placeholders are replaced with the respective set of parameters.

The << *operator* is the output operator and transfers Tasklet results from C-- back to the consumer application. The output operator attaches a result and its explicit data type information to the Tasklet result message, which is transmitted to the host application after the Tasklet is terminated. The structure of a language is defined by its grammar (or productions), which can be found in Appendix A.

5.3.5. Execution Layer

The execution environment of Tasklets consists of a TVM and a TVM Manager, which orchestrates all TVM instances on a host. The TVM Manager further allocates incoming Tasklets to idle TVMs.

Tasklet Virtual Machine

There are two options to design an environment for the execution of Tasklets: using an existing execution environment for Tasklets or developing a virtual machine from scratch. There is a multitude of existing virtual machines, which can potentially be used for Tasklet execution. For example Google's V8[2], which runs on several operations system and executes JavaScript code. However, this entails several issues: Instead of having a lightweight intermediate byte code format, heavy JavaScript source code is shipped between consumers and providers. Further, all local system resources can be accessed by the V8 engine, which implicates security issues when executing unknown code. Other open source solutions are JamVM[3] or Avian[4]. Both are based on Java, lightweight, and run on multiple platforms. However, adaptability, extensibility, and the isolation from system resources are issues of these solutions. Contrary to that, developing a virtual machine from scratch can solve that at the cost of development effort and restricted execution performance. Especially for requirements like QoC algorithms, unobtrusiveness for the user, and migration, flexibility and adaptability are vital. Therefore, the *Tasklet Virtual Machine* (TVM) is designed and developed from scratch with the focus on adaptability and extensibility.

The TVM provides a homogeneous abstraction for otherwise heterogeneous resources. It encompasses a stack-based bytecode interpreter that holds all temporary values on an operand stack without using any general-purpose registers. These temporary values include arithmetic operands, parameters, memory references, and result values. The memory of the TVM has four different segments for the stack, the program text, the constant pool, and the heap. The program text segment stores the bytecode of the current Tasklet and the constant pool all

[2]Google's V8: https://opensource.google.com/projects/v8/, accessed: 10/01/2019
[3]JamVM: http://jamvm.sourceforge.net/, accessed: 10/01/2019
[4]Avian: https://github.com/ReadyTalk/avian, accessed: 10/01/2019

constant values, as well as the parameters transferred from the host language. Both segments are read-only. Similar to the stack segment, the heap segment changes in size during runtime. The heap space stores all dynamic data that does not fit on the stack, including arrays of all kind. Further, the TVM holds special registers for the program counter, the base address, and a pointer to the top of the stack.

The TVM is built to be lightweight with a minimal memory footprint of about 400 KB in idle state in order to run on thin clients. TVMs are single threaded processes and do not support multi-threading themselves. They sequentially execute incoming Tasklets without interruptions, similar to batch systems. However, the resource owner may terminate a TVM at any time, for example in an excess capacity scenario, where users contribute resources as long as they do not need them locally. TVMs do not support system calls or access to any system resources from the Tasklet logic written in C--. To achieve isolation from other system components, each TVM runs in a separate process. During execution, code, parameters, and data of a Tasklet never leave the volatile memory of a physical machine. For communication, TVMs have two channels. The first for incoming Tasklets and outgoing results, and the second one for administrative messages, such as Tasklet cancellation. The error handling of the TVM is straightforward and in best-effort manner. If a runtime error occurs, the execution is terminated and the Tasklet is dropped. QoC goals, as reliability, are handled in the upper layers of the middleware. Finally, the TVM resets itself entirely after each Tasklet execution and does not store any state.

Two options exist, on which abstraction level the TVM can be executed: First, in the application level of the operating system. This is the standard way, where the TVM is compiled as a process for the respective platform and scheduled by the underlying operating system. Second, the TVM is integrated into the operating system kernel and runs on a dedicated processing core. This avoids the overhead of the operating system scheduler and applies a non-preemptive scheduling mechanism as long as resources are available. In case the user requires the system resource, all Tasklet executions are stopped. This approach is more performant, but requires high development effort, as well as customization for each operating system. Further, the operating system must be available in source code to extend it with the Tasklet execution environment.

TVM Manager

On each resource provider, a *TVM manager* handles the lifecycles of all TVMs that run locally on the respective device. It starts and stops the TVMs according to the device's context, including the number of physical processing cores and the current workload of the system. In the default setting, the TVM manager starts one TVM per processing core and schedules Tasklets preferably on idle TVMs. In case of excess capacities where users share their private resources, the TVM manager terminates or pauses TVMs when the resources are needed locally. This avoids the user to be disturbed by the Tasklet system. Ideally, the user does not even recognize that Tasklets are executed. Further, the TVM manager monitors node failures, the Tasklet throughput, and turnaround times. Additionally, the TVM manager provides information about the state of each individual TVM for the system's resource management.

5.3.6. Distribution Layer

So far, the thesis introduced the construction and execution of Tasklets. To allocate Tasklets in the system remotely, the distribution layer combines an orchestration with a broker network overlay. Furthermore, it manages the Tasklet and result exchange among all instances in the system and federates all local TVMs through the TVM manager. The orchestration schedules Tasklets on local or remote TVMs. It therefore requests available resources from the resource broker and forwards Tasklets for execution. The orchestration of the executing instance sends the results to the orchestration on the device of the user application. An application protocol supports the structured message exchange. Further, the orchestration enforces QoC goals, which are stated by the application developer. As a result, QoC extends the otherwise best-effort execution of Tasklets with various guarantees.

The resources in the system, in other words, the TVMs, are globally managed by the resource brokers. The initial bootstrapping in the system is realized in a mediator-based fashion. Consumers and providers receive a list of brokers, ping them, and connect to the one with the shortest round trip time (RTT). For load balancing and request forwarding, the brokers themselves are organized in a peer-to-peer overlay. Each broker has a target size of managed TVMs, with an upper bound to assure responsiveness and a lower bound to be able to provide sufficient resources for consumers. Brokers can merge or split up their pools of providers to deal with fluctuation in the system. Providers can spawn zero to many TVMs on startup, as well as start or stop TVMs at runtime. Each TVM registers itself at its broker and de-registers once it is terminated via the TVM manager. A heartbeat channel detects implicit leaves of providers. Further, it informs the broker about current Tasklet executions. Thus, the brokers have consistent information about all resource providers.

Quality of Computation

The main idea of the QoC concept is that one generic underlying system can be used by a large variety of applications. For some applications, a best-effort execution is appropriate, but for the majority further qualities are required. Therefore, the QoC layer is introduced to allow developers to tailor the level of computation quality to each application individually. They can define fine-granular execution requirements for each computationally intensive part of an application with negligible additional programming effort. This results in the flexibility to request different guarantees in the same application. The QoC concept is further divided into two parts: the QoC goals and the QoC mechanisms. A QoC goal represents the type and the level of a particular execution quality for the application developer regardless of its enforcement. The QoC mechanisms, on the other hand, represent a particular mode of execution, that can support different QoC goals. Hence, a developer sets a QoC goal and the middleware uses QoC mechanisms to enforce it. For example, if a developer wants a fast execution for Tasklets, the *Speed QoC* can be applied. For its enforcement, the middleware decides on an mechanism that fits best, depending on the current system state. Two options are: (1) scheduling multiple copies of a Tasklet and using the first arriving result or (2) selecting the fastest available resource provider. The first option does not only improve the execution speed, but also increases the reliability of the execution. In edge environments, devices fluctuate constantly and may leave the system ungracefully, which leads to execution failures. Therefore, $n + 1$ copies of the same Tasklet are able to cope with n failures. This example shows that a QoC goal can be enforced by multiple mechanisms and a mechanism can enforce multiple goals.

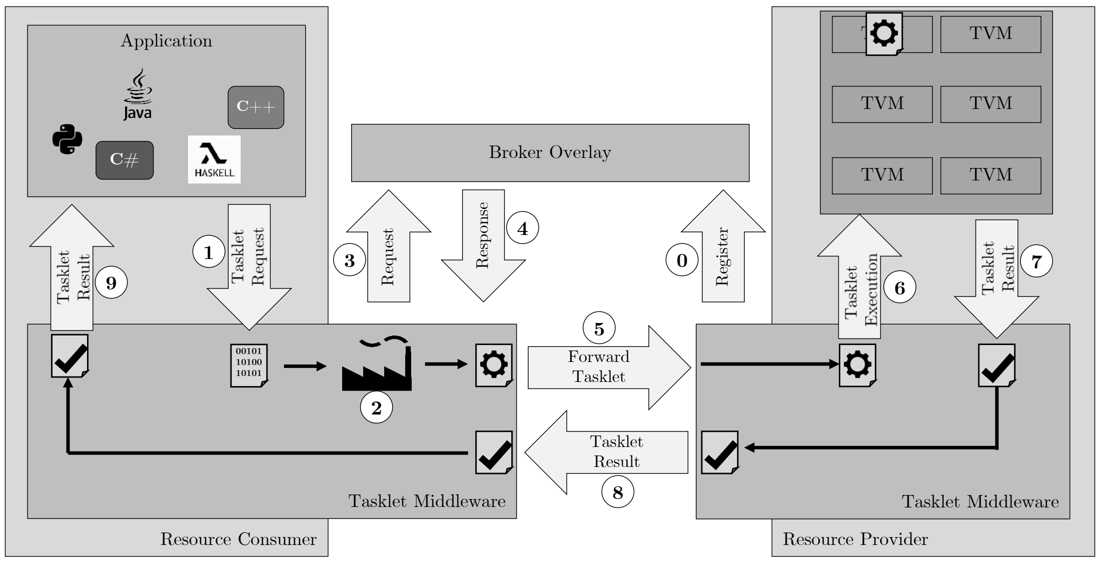


Figure 5.9.: Lifecycle of a Tasklet execution. For the execution, the Tasklet middleware on the consumer side prepares the Tasklet and transfers it to the providers side. After the execution, the results are forwarded to the consumer application. The numbers (0-9) indicate all execution steps.

To augment the best-effort execution that the Tasklet system offers, various QoC goals are required. These goals are aligned with the requirements of the application model from Section 5.2. The goals are *reliability*, *speed*, *precision*, *privacy*, *cost*, and *energy*. The enforcement of these goals is taken over by various QoC mechanisms, like *multiple execution*, *strong distribution*, *local execution*, and *retransmission*. Both, QoC goals and mechanisms are out of the focus of this thesis. More detail about QoC can be found in [166][5].

5.3.7. Tasklet Lifecycle

After the description of all core components of the Tasklet system, the Tasklet lifecycle is introduced. The lifecycle is illustrated in Figure 5.9 and each step is marked with a number. The Tasklet application initiates a Tasklet request by calling the respective API functions. After that, the plain Tasklet is passed on to the middleware (step 1). The factory generates the bytecode and assembles the Tasklet closure (step 2). Then, the factory forwards the Tasklet to the orchestration, which assigns the Tasklet an identification handle. Further, the

[5][166] is joint work with J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker

QoC parameters are extracted and analyzed. Based on that, the orchestration prepares a resource request and sends it to the responsible resource broker (step 3). The resource broker runs matching algorithms that consider the stated QoC parameters and the current state of all resources in the system. With the results of that allocation process, a response message is prepared and sent back to the orchestration of the consumer (step 4). The consumer retrieves the messages and the orchestration forwards the Tasklet to one or multiple assigned resource providers (step 5).

After the provider's orchestration receives the Tasklet, it forwards it to a free TVM (step 6). On the TVM, the Tasklet is demarshalled and the stack machine as well as the memory segments are filled with its data. Next, the TVM starts the execution. In case of failures or aborts, the TVM stops the execution and dedicates the handling to the orchestration layer. After the execution successfully terminates, the TVM assembles a result message, which the system finally forwards to the Tasklet application via the orchestration (step 7-9).

5.3.8. Performance Measure

This section is based on [164][6]. The performance of the distributed computing system can be measured in multiple ways. For providers, the average utilization and the throughput are important measures. The average utilization is the work-idle-ratio of a provider in a certain time window and the throughput is the number of successfully executed Tasklets per time. For consumers, the cost of executing Tasklets should be minimized. Cost, in this case, can refer to either energy consumption, turnaround time, or monetary compensation for providers. This thesis focuses the overall turnaround time of a Tasklet as a measure of performance, since it reflects the experienced quality for consumers and the potential throughput for providers.

The *total turnaround time* (T) to execute a Tasklet is the sum of the scheduling time (S), the time for computation on the TVM (C), and the time to send the result back to the consumer (R). It can be stated as follows:

$$T = S + C + R \tag{5.1}$$

[6] [164] is joint work with J. Edinger, M. Breitbach, and C. Becker

The *scheduling time* describes the time to offload the Tasklet to a remote provider. Initially, the offloading process starts with a resource request from the consumer to the broker. The broker selects a suitable provider and returns the information to the consumer. Subsequently, the consumer forwards the Tasklet to the provider. The time depends on the size of the Tasklet, the network characteristics, and the performance of the scheduling algorithm. The *computation time* measures the time difference between the beginning and the end of the execution on the TVM. Thus, the time depends on the computational effort of the Tasklet, which can be measured in number of instructions that have to be executed and the performance of the underlying hardware. The *result handling time* states the required time to forward the result back to the consumer, which depends on the result size and the current network state.

The turnaround time of T in Equation 5.1 holds under the assumption that no faults occur. That implies that the offloading process and result handling happen exactly once and that the computation is performed without any interruptions. This assumption does not hold in unpredictable edge computing environments where faults and device fluctuation cannot be avoided. The extended Equation 5.2 considers these faults and calculates the required time when n faults occur.

$$T = S + \sum_{i=0}^{n}(D_i + RS_i + C'_i) + C + R \tag{5.2}$$

$$C_{effective} \leq \sum_{i=0}^{n}(C'_i) + C'' \tag{5.3}$$

The *scheduling time* (S), computation time (C), and the result handling time (R) are similar to the values in the simplified Equation 5.1. Additionally, the term in the sum operator measures the delay that is caused by n faults. It consists of the time required to detect the fault (D), the time required for rescheduling the Tasklet (RS), and the time spent for each computation attempt (C'). In terms of computation time, Equation 5.3 describes the relation between the final computation time (C''), the sum of all lost computations (C'_i), and the effective computation time ($C_{effective}$). This holds for all unpredicted system faults. In case of a graceful system leave of a provider, the detection time (D) is equal to zero.

$$T_{bundle} = \max_{1 \leq i \leq n} (T_i) \tag{5.4}$$

So far, T is considered as the overall execution time of a single Tasklet. To distribute the computational load, a task can be split into multiple Tasklets, which run in parallel. The execution time of a bundle of Tasklets (T_{bundle}) is shown in Equation 5.4 and indicates that the Tasklet with the longest execution determines the bundle execution time. Especially in edge environments, this can lead to bottlenecks which become larger due to device performance and task heterogeneity.

Based on the design of the Tasklet system, the edge support layer is now introduced to cope with the characteristics of user-controlled devices.

5.4. System Support for Edge Environments

The challenges in edge environments are platform heterogeneity, task requirements, fluctuation, failures, performance heterogeneity, and irregular task structures. This section introduces the *edge support layer* that handles these challenges subsequently. First, based on virtualization and specific architectures, the system copes with platform heterogeneity. Second, by combining cloud and edge environments, task requirements regarding latency and performance are fulfilled. To scale-up the performance of the edge, the parallelization degree is increased. Therefore, a broader range of devices is harnessed. This may include less performant and more unstable devices, which leads to a higher device fluctuation and more execution failures. Thus, reactive and proactive migration is introduced to reduce the impacts of fluctuation and failures. Further, the probability for execution bottlenecks is increased as well. Bottlenecks can have two sources: slow resource providers and irregular tasks structures. Two algorithms are introduced to cope with both problems: performance-aware partitioning and microtasking. Performance-aware microtasking is a combination of both approaches and handles bottlenecks that are caused by both issues.

5.4.1. Platform Heterogeneity

The core of the Tasklet system copes with a certain level of platform heterogeneity already. Based on the TVM, various operating systems, like Windows, MacOS, and Linux, as well as desktop computing architectures are integrated. In edge and IoT environments, the platform heterogeneity is higher than in traditional cloud, cluster, or grid systems. Especially mobile devices, microcontrollers, and GPUs introduce architectures that are different from standard computing hardware. Further, these kind of devices are prominent in the modern computing landscape. Mobile devices and microcontrollers use system on a chip architectures, which are integrated circuits that include all components of a computer, like CPU, memory, buses, interfaces, and a GPU. GPUs are specialized processing units, which are optimized for highly parallel graphics rendering. However, with general purpose programming models for GPUs, like OpenCL [177], the scope of GPUs is extended. This section presents the integration of mobile devices, microcontrollers, and GPUs into the Tasklet system.

Mobile Tasklets – Integration of Mobile Devices

Mobile devices are ubiquitous, but their resources are limited. They must be capable to run computationally intensive software, for example for image stitching, face recognition, and simulation-based artificial intelligence. As a solution, mobile devices can use computation placement systems to increase the performance of applications. This section presents the design of *Mobile Tasklets* [163][7] and its overall architecture. To use mobile devices in a distributed computing environment, special challenges must be tackled. These challenges are attributable to the architecture and mobile behavior of these devices. Some examples for that are context dynamism, heterogeneity, faults, network connection, and energy efficiency.

Architecture: The architecture of Mobile Tasklets has two layers and is shown in Figure 5.10. The layers are designed to overcome the issues with mobile devices by considering their context. The bottom layer is the service application that consists of the Tasklet core components and a service wrapper. It interacts with the plain resources of the mobile device and is a standalone application. When

[7][163] is joint work with J. Edinger, T. Borlinghaus, J. M. Paluska, and C. Becker

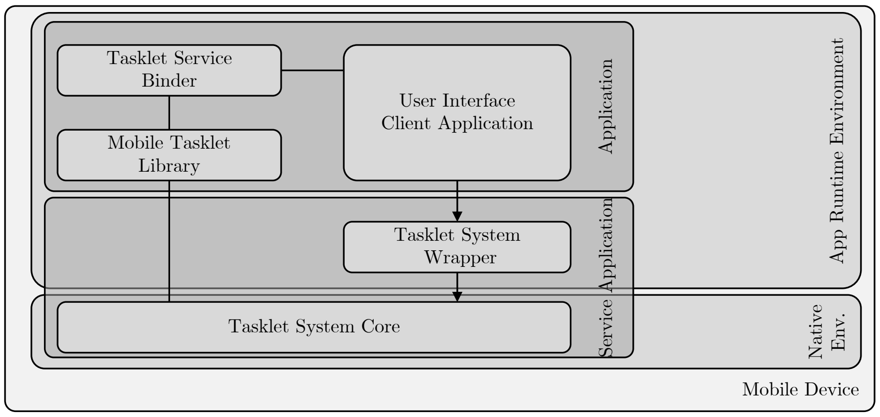


Figure 5.10.: Overall architecture of the Mobile Tasklet approach. On the upper layer, the consumer application interacts with the user and makes use of Tasklets. It runs the Mobile Tasklet library and a component for binding the Tasklet service. The lower layer of the system runs the service application and contains the Tasklet system core and the Tasklet system wrapper.

the service application is started, the device may provide its resources to others. The unmodified C code of the Tasklet core is executed in the native execution environment of the mobile device. The Tasklet system wrapper closes the gap between the host language environment and native execution environment of the device. The top layer consists of the consumer application itself, the Tasklet service binder, and the Mobile Tasklet library. The application represents all kind of mobile applications that benefit from code offloading. It integrates the Tasklet service binder, which facilitates means of communication to the lower layer. To integrate the Tasklet system, the application includes the Mobile library, analog to the standard Tasklet library. All requests to the library done via the Tasklet binder.

To go one step further, the integration of even smaller devices is introduced next.

The Tasklet Gateway – Integration of Thin Devices

Tasklets are designed to be initiated by almost any device. However, creating and executing Tasklets is problematic for devices that do not support the execution of the Tasklet runtime environment, as well as for embedded systems with very limited resources, such as sensors, actuators, or microcontrollers. The tasks on the resource consumer side regarding Tasklet compilation, assembly, scheduling, and result handling assume computational capabilities that these thin devices often

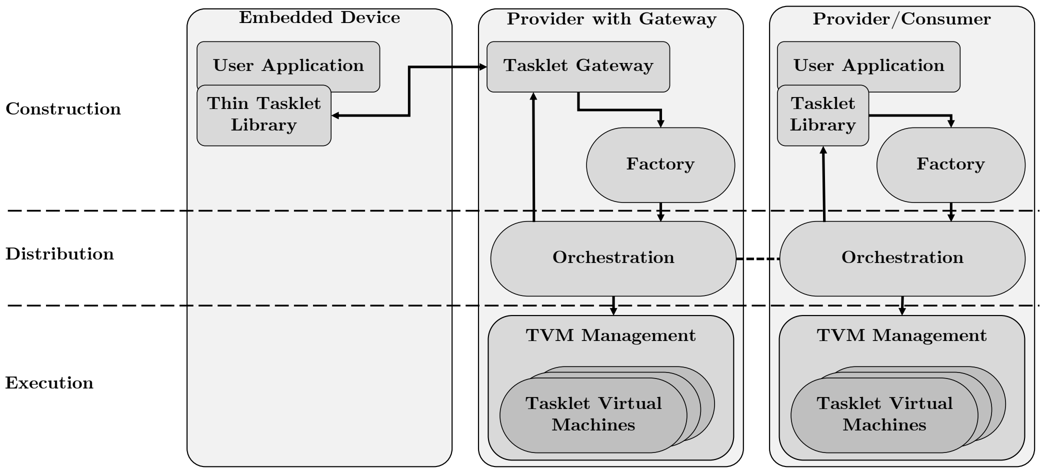


Figure 5.11.: Architecture of the Tasklet gateway. The embedded device runs the consumer application that uses the thin Tasklet library. This library is connected to the Tasklet gateway that takes over the majority of the tasks. The gateway acts as an application proxy: it forwards the plain Tasklet request to the middleware and returns the execution results back to the embedded device.

not have. Further, especially scheduling can become tedious for mobile devices that frequently change their network connectivity. They might either change their IP address or disconnect from the network entirely, resulting in lost result messages, as the executing provider would try to send the result to a deprecated IP address.

In such settings, a separation of application and Tasklet middleware is desirable. For this, the *Tasklet gateway* in combination with the *thin Tasklet library* is added to the system's landscape. The Tasklet gateway reduces the required effort for Tasklet submission to an absolute minimum and serves as a reliable and stable endpoint for the connection to the executing instance. The thin Tasklet library is tailored to the characteristics of resource-limited devices. From the application perspective, the thin library is similar to the standard Tasklet library. In addition, it connects the application to the gateway and offers functionality to find a gateway device, manage the gateway connection, and reduce the network traffic for Tasklet requests.

Figure 5.11 shows the architecture of the approach. The gateway runs on dedicated resource providers and connects applications running on thin clients in its nearby environment to the Tasklet system. The application sends the same request to the gateway as it would send to a local middleware. One major difference is that the Tasklet request is sent over a network to the gateway. This can be a local network or the Internet.

On the dedicated resource provider, the gateway acts as an application towards its local middleware. That is, it manipulates the Tasklet request message and pretends to be the initiator of the Tasklet. Consequently, the middleware treats the request like any other request from a local application, and the executing instance sends the results back to the respective gateway. The gateway, again, manipulates the information in the result message and forwards it to the actual application on the thin client. One Tasklet gateway is able to serve multiple thin devices. To achieve this, each device is uniquely identified and requests are handled separately. In case multiple Tasklets are started as a bundle, the gateway gathers the results and transfers them efficiently. The thin Tasklet library in combination with the gateway provide thin devices, like sensors, microprocessors, and IoT devices, the use of the Tasklet system.

Next, the integration of GPUs in the Tasklet system landscape is presented.

The GTVM – GPU-Accelerated Tasklet Execution

In edge computing systems, computation is rather offloaded to nearby resources than to the cloud due to latency reasons. However, the performance demand in the edge grows steadily, which makes nearby resources insufficient for many applications. Additionally, the amount of parallel tasks in the edge increases, based on trends like machine learning, Internet of Things, and artificial intelligence. The trade-off between high performance of the cloud and low latency of the edge has to be considered for the scheduling. Many edge devices have powerful co-processors in form of their graphics processing unit (GPU), which are mostly unused. These processing units have specialized parallel architectures, which are, different from standard CPUs, rather complex to use. Exploiting GPUs increases the performance of edge devices drastically. This section presents the design of GPU-accelerated task execution [161][8] for edge computing environments.

[8][161] is joint work with J. Edinger, and C. Becker

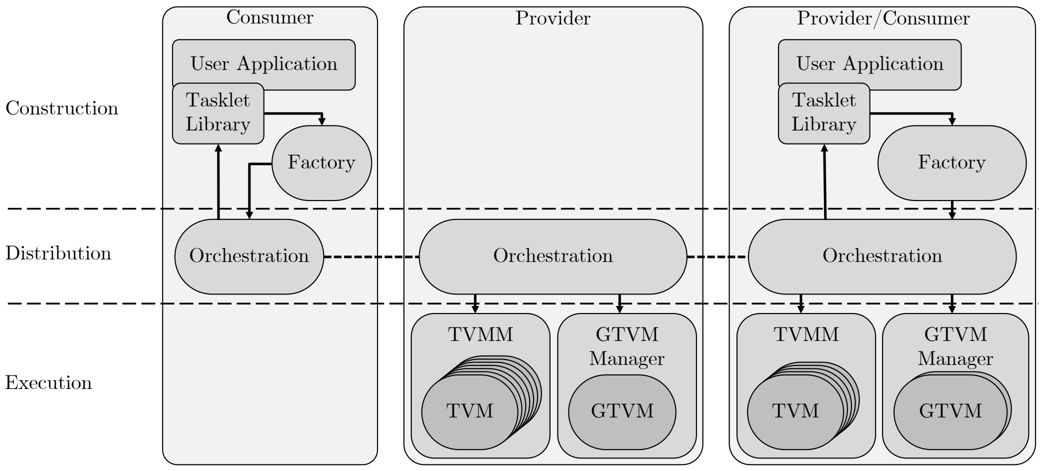


Figure 5.12.: The Tasklet system architecture with GPU extension. The construction layer generates Tasklet request and consists of the Tasklet factory and library. The distribution layer uses the Tasklet orchestration to allocate Tasklets remotely. The execution layer contains Tasklet virtual machines, which run on heterogeneous devices. The new system extension uses the GTVM manager and the GTVM to support GPU-accelerated Tasklet execution.

The design consists of two parts: First, an orchestration module for GPU resources on the provider and, second, a specialized version of the TVM – the so called *GPU-based Tasklet Virtual Machine* (GTVM). The overall objectives of the design include parallelization, overcoming heterogeneity, and unobtrusiveness. The standard TVM does not support any kind of parallelism. When application programmers want to make use of parallel execution, they have to split the task up on application level and initiate multiple Tasklets. In contrast to that, the GTVM utilizes the parallel architecture of GPUs. Since there are different GPUs with diverse architectures on the market, the design has to cope with heterogeneity. The last design focus is unobtrusiveness for the user. The main responsibility of the GPU is rendering the graphical user interface (GUI) for the user, which should not be compromised by the design. Figure 5.12 shows the integration of the new components. Each provider has a *GTVM manager* and one *GTVM* per installed GPU. Depending on the task size, the memory consumption, and the current workload, each GTVM starts multiple *Tasklet threads*. Next, the three main design characteristics are described.

Parallelization: The standard TVM assumes the process isolation of the operating system to protect Tasklets from mutual violation and malicious behavior. However, the memory allocation on GPUs works differently. All running programs can access the GPU memory. Thus, there is no natural process isolation. For that reason, the architecture supports two different GPU execution modes. Both modes do not allow different Tasklets on the same GTVM concurrently. The first mode executes multiple instances of the same Tasklet in parallel. The second mode executes the same Tasklet several times, but with different execution parameters. Therefore, the Tasklet is forwarded to the GPU together with a set of parameters.

Heterogeneity: GPUs are highly heterogeneous regarding their architecture, number of processing entities, and memory structure. The GTVM overcomes these heterogeneities in proposing a design that copes with diverse GPU models. Therefore, OpenCL is used, which enables to compile the same GTVM version to various GPUs. This comes at cost of overhead that emerges from running non-native GPU code. Further, systems with multiple GPUs are considered. The GTVM manager optimizes the scheduling strategy and considers different GPU characteristics. Besides, the GTVM copes with different kinds of operating systems.

Unobtrusiveness: The main task of the GPU is rendering the GUI. Compared to a CPU, a GPU is scheduled in a non-preemptive manner, which makes multiprogramming complex. Thus, while the GPU is used as a co-processor (e.g., to run Tasklets) the operating system cannot render the GUI. In this case, the Tasklet execution interferes with the user through judder effects or even a blocked GUI. The present approach stops the Tasklet execution on the GPU to yield the execution periodically. To realize that, a snapshot is created and used to resume the Tasklet execution after the operating system rendered the GUI. Hence, no Tasklet progress is lost and the user is not disturbed. In multi-GPU systems, one GPU renders the GUI while other GPUs are dedicated for Tasklet execution.

After the three platform integrations, scheduling is the focus of the following section. Especially the role of cloud in combination with nearby resources is presented. Therefore, a hybrid scheduling is introduced that uses remote cloud and ad-hoc edge computing resources.

5.4.2. Hybrid Tasklet Scheduling

MCC enables mobile devices to augment their computational capabilities with powerful centralized resources. However, due to latency issues, MCC is unsuitable in many situations. Edge computing appeared as a more decentralized paradigm, which utilizes resource consumers that are in the proximity of the consumer device. It benefits from the continuously increasing amount as well as the enhancing performance of end-user devices and their network performance. Compared to MCC, nearby resources are limited, but reachable with a substantially shorter latency. Therefore, a combination of both approaches can drastically improve the performance of mobile applications. Based on the introduced application model, applications that are executed in the edge can have two specific requirements: performance and latency. In case of long-running, complex executions, it is required to have a high performance resource for execution. Other applications require a timely execution with a low latency.

This section introduces a hybrid scheduling approach in edge and cloud environments [165][9]. It combines the benefits of resource-rich cloud computing with the high responsiveness of edge computing. In addition to the centralized scheduling on remote edge and cloud resources, an ad-hoc scheduling mechanism for nearby edge environments is introduced. Depending on the device context and the task structure the system decides between (i) powerful and stable cloud resources, (ii) low latency and lightweight edge resources, or (iii) powerful and low-priced remote edge resources, as shown in Figure 5.13.

Considering the diverse characteristics of edge and cloud devices, the scheduling process of Tasklets is rather complex. For the execution on remote resources, the system uses centralized scheduling via the broker. This implies one RTT for the request from the consumer to the broker and one RTT between the consumer and the provider for the Tasklet and results exchange. On the contrary, for nearby edge resources, the Tasklet is allocated in an ad-hoc manner with a direct message. However, due to their movement, mobile devices can have high churn rates. This increases the error rate, since a Tasklet result is lost, if the consumer or provider is not in range anymore. Moreover, the performance as well as the power supply of mobile devices are limited. To exploit the benefits of cloud, remote edge, and nearby edge resource, a hybrid architecture is proposed.

[9][165] is joint work with J. Edinger, J. Eckrich, M. Breitbach, and C. Becker

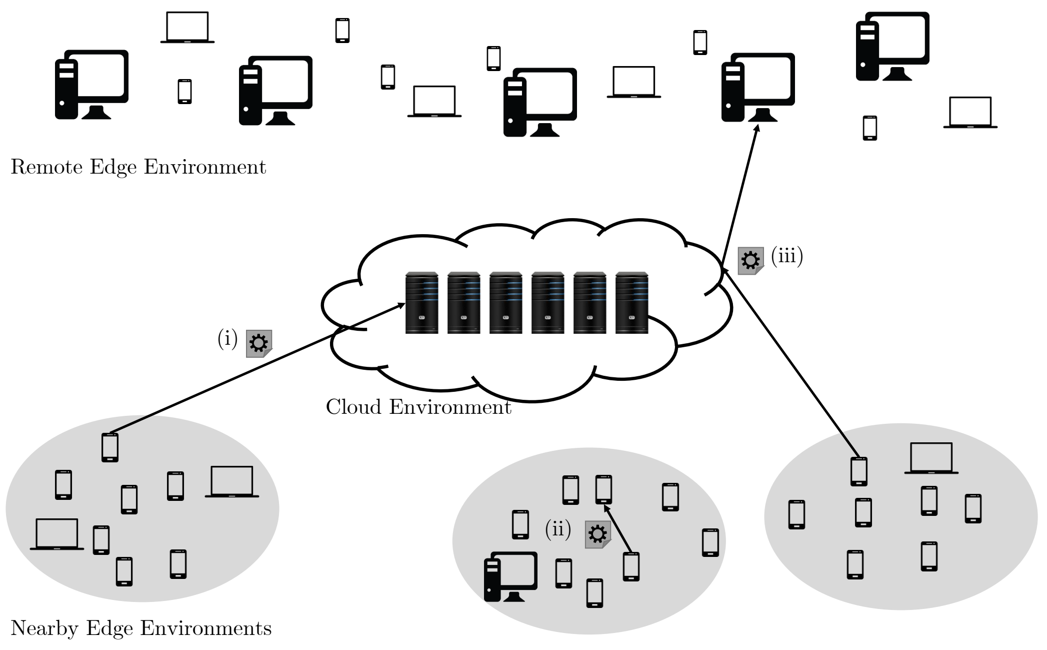


Figure 5.13.: The hybrid scheduling model. The three layered model combines (i) cloud resources with (ii) local and (iii) remote edge resources.

Cloud resources are stable and reliable, since the respective cloud instances are designated to execute Tasklets. The scheduling is centralized within the broker network. Cloud resources are especially suitable for long-running tasks with a high priority, which require high execution performance. However, scheduling latencies are involved and these resources are more expensive.

Remote edge resources are similar to the cloud regarding the scheduling latency. Further, their execution speed is similar to the cloud, since full-fledged computers with high performances can be selected with QoC. Nevertheless, remote edge resources strongly differ in terms of stability and reliability, since they can leave the system at any time. Again, long-running Tasklets, which require heavy computation are suitable for this kind of resource, but without high priorities, reliability, or dependencies.

Nearby edge resources are spontaneously-connected groups of devices in proximity. They have a very short communication delay, but also limited execution performance. This resource is especially suitable for bursts of short Tasklets, which require high responsiveness.

The design of the hybrid scheduling approach is twofold: On the one hand it consists of the centralized scheduling system of the standard Tasklet system, the so called remote resource broker. On the other hand a decentralized ad-hoc scheduling for Tasklets is introduced. With this approach, mobile devices can spontaneously form groups that exchange Tasklets with low latencies and without involving centralized cloud resources. This set of features is realized in the *local broker* component. Depending on the situation, the Tasklet system can use the local or the remote scheduling system.

Since Tasklet allocation on remote resources is a part of the core system, this section focuses on the nearby edge scheduling. To achieve this, the Mobile Tasklet architecture is extended with the local broker and a component that decides on the most suitable resource type for each Tasklet. This leads to different scheduling strategies: in case of short and highly responsive Tasklets, the system schedules within the nearby edge. In contrast, an important long-running Tasklet is scheduled via the remote resource broker to a stable and powerful cloud resource. Other long-running Tasklets that are not as critical are scheduled to remote edge devices. Not only the Tasklet characteristics are crucial for the scheduling decision, but also the environment and the device itself. In case the nearby environment has a high churn rate or the device itself is moving, the probability for a remote scheduling approach is high. Next, the design of the local broker is explained in detail.

Scheduling in the Edge

Figure 5.14 presents the system model of the local broker. Each participating device runs the local broker component. It creates and manages local groups of devices and provides the same service as the centralized broker of the Tasklet system. Within the group, context information is exchanged to allow context-aware Tasklet allocation. The local broker is designed as an extension of the Mobile Tasklets approach. It adds three components to the architecture: the *Tasklet connector*, the *context engine*, and the *network handler. The Tasklet connector* implements the system integration of the Tasklet core system and the new local broker component. *The context engine* reasons internal context of a device. Further, it collects external context information of nearby devices by

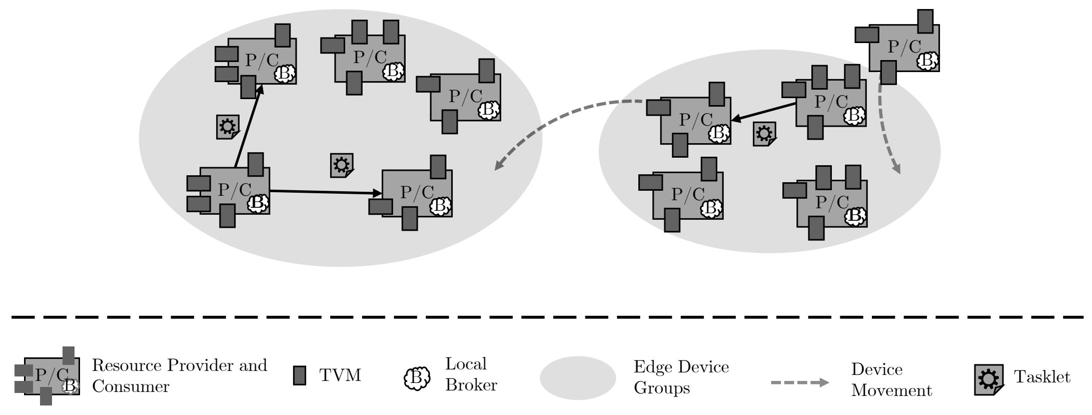


Figure 5.14.: The Local Broker system model. Decentralized scheduling with Tasklets in spontaneously-formed groups of edge devices, each of which has one group owner (marked in bold).

means of which it calculates a utility ranking for Tasklet allocations. *The network handler* establishes spontaneous connections between nearby devices. Therefore, it uses a network underlay, which utilizes peer-to-peer technology and a network overlay that is used for Tasklet allocations. Next, the *context engine* and the influence of context are explained in more detail.

Context-awareness For the nearby scheduling of Tasklets, the device's context is considered to improve the execution quality. This is done by the so called *context engine* and independent from the QoC goals that are set by the developer. The context currently encompasses the CPU utilization, the CPU temperature, the battery level and status, as well as the Wi-Fi signal strength. *The battery* is an important factor. It determines whether a device is charging and what the current battery level is. *The Wi-Fi signal strength* has an impact on the data transfer rate between the participating devices. Further, a decreasing signal strength may indicate that a device will leave the group soon. The *CPU utilization and temperature* is balanced by the system among all devices of a group. Since smartphones regulate the CPU temperature by adjusting the CPU clock, the temperature is also important. A device that is charging and exposed to direct sunlight, has a high temperature without doing heavy computation. Therefore, the clock speed is reduced. Based on these context values, a utility function is used to calculate a utility value for each device. Each context value c_i is normalized and multiplied with corresponding weight w_i to calculate the utility value U_i for a device:

$$U_i = w_1 * c_1 + w_2 * c_2 + \ldots + w_n * c_n \tag{5.5}$$

Additionally, the context engine has a set of rules. An example rule set could be, that the battery must be above 10% and the CPU temperature below 60°C to execute a Tasklet. In case one of the rules is infringed, the devices obtains an utility value U_i of 0. This mechanism establishes load balancing among the participating devices and increases the execution reliability. The context engine collects the external context and the local context of the device itself. This process is done periodically, depending on the changing of context values. When the alternation rate is high, the intervals are shorter and vice versa.

Based on that design, a hybrid scheduling is realized. Next, the performance increase of edge and cloud resources is further examined.

5.4.3. Enhancing Performance of Low Latency Edge Resources

So far, the computing landscape of the Tasklet system is extended with typical edge devices. The hybrid scheduling approach demonstrates the ability to use low-latency edge and high-performance cloud resources in one system. The subsequent question is, how can the performance in the edge be improved further?

The dark gray space in Figure 5.15 marks an optimal environment regarding performance and latency. To approach it, two options exist. First, the latency of cloud environments can be reduced, which has been done by fog computing approaches and most prominently by Cloudlets [158]. Generally, the idea is to distribute cloud infrastructure and deploy them in proximity of the edge.

The second approach is to increase the performance of the edge. Edge resource providers, however, are limited in number and their individual serial execution performance. A first step in that direction was the integration of GPUs, which is, however, limited to a certain kind of application that benefits from highly parallel architectures. Thus, scaling up the resources is not straightforward, but can be solved with a higher degree of parallelization by accumulating multiple resource

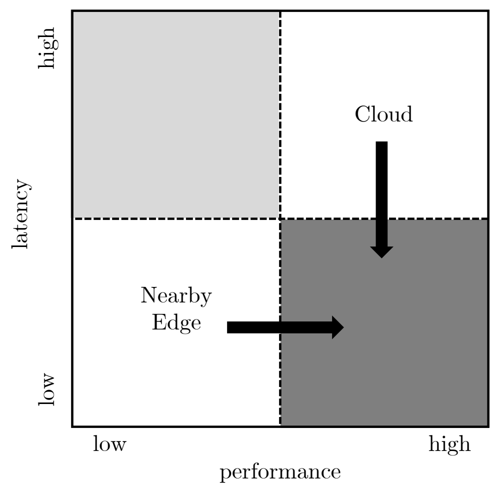


Figure 5.15.: Improvement potential for Edge and Cloud resources. Edge resources have low latencies, but are limited in performance. Cloud resources, on the other hand, are very powerful, but introduce higher communication latencies. The dark gray area marks the target space that can be reached by reducing the latency of the cloud or increasing the performance of the edge.

providers. This applies only for tasks that can be fully parallelized. The static partitioning of a task done by the application developer before runtime is not adjustable and does not consider any runtime information. Thus, tailoring the task partitioning to the current state of the environment enables to dynamically decide on an optimal strategy. As a solution, automatic data level partitioning is introduced. The application developer specifies the smallest granularity a task can be split in and information about the task structure. The middleware applies this information in combination with the number of available resource providers to split the task dynamically during runtime. As a result, the middleware can find the optimal strategy for distributing the workload to the nearby edge environment. Based on that, a large amount of computational performance can be accumulated. However, this approach has two major drawbacks that increase the response times drastically: faults and bottlenecks.

Especially in edge environments, device fluctuation and unreliable task execution can occur frequently and generate large execution delays. To cope with this problem and increase the fault tolerance, two task migration algorithms are introduced in Section 5.4.4. The second problem are bottlenecks, which can

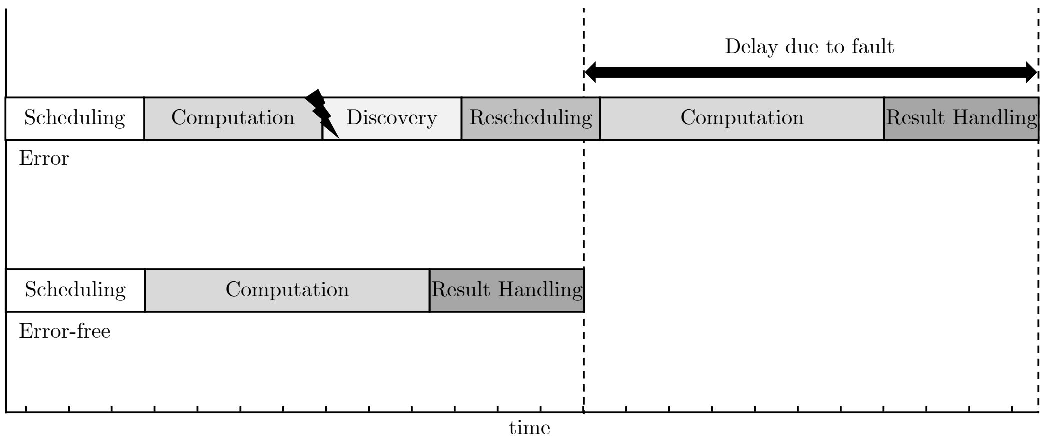


Figure 5.16.: The impact of execution failures. Error free (bottom) versus an erroneous (top) remote task execution. The black line indicates the delay caused by the error.

emerge from device heterogeneity and irregular task structures. Section 5.4.5 introduces algorithms that utilize environment information in combination with risk distribution strategies to handle both sources of bottlenecks.

5.4.4. Task Migration for Fault Tolerance

Figure 5.16 shows the impact of failures in comparison with an error free execution. Analogous to Equation 5.2, for each failure occurrence, the times for failure discovery, rescheduling, and redundant computation are added to the overall execution time. To reduce the overall response time of Tasklets, four options for the improvement of fault tolerance are feasible. First, the number of faults n can be reduced by selecting the resource provider based on historic behavior. This algorithm is called fault avoidance, is published in [63][10], and focuses on remote desktop grid environments. However, fault avoidance may restrict the number of potential providers and is not optimized for nearby edge environments. In environments where resources are already restricted in number, filtering reduces the allocatable resources to a minimum. The second solution is to reduce the discovery time D, which can be done by increasing the heartbeat rate and reducing the timeout. This approach, however, increases the overhead and may lead to unwanted parallel executions of the same task. Third, the time for rescheduling RS after a fault occurred can be reduced. Therefore, decentralized scheduling

[10][63] is joint work with J. Edinger, C. Krupitzer, V. Raychoudhury, and C. Becker

based on cache lists is proposed in [61][11]. It minimizes the communication with the central broker to accelerate the resource requests. However, this solution is only optimized for desktop grid environments and not for edge computing. Further, the scheduling in the nearby edge works without any centralized entity, as described in Section 5.4.2. The fourth option for improvement is to reduce the redundant computation C'. In case of an execution fault, the computation does not start from the beginning, but continues based on an intermediate execution state. As a solution, this section introduces two algorithms for task migration to cope with implicit and explicit faults. This approach is based on [164][12].

An explicit system leave implies that a resource provider executing a task leaves the system with notice. Therefore, the middlware can gracefully transfer the computation to another provider. Explicit leaves can happen when the network connection becomes worse, the battery state changes, or the user closes the middleware. In contrast, an implicit leave is not foreseeable. The provider stops all task execution and opts out without any notice. Implicit leaves arise when the user shuts down the device, forcibly exists the middleware, or suddenly loses the network connection. The following migration algorithms cope with both types of leaves.

Task migration describes the process of recording and transferring an execution state from one resource provider to another in order to continue execution of the task there. The Tasklet system uses the TVMs to provide a homogeneous runtime environment on all participating devices. The state of these TVMs is recorded in a snapshot, which for example includes the state of the stack, heap, and the program counter. Furthermore, intermediate results of the task execution are recorded. After each instruction of a TVM, a provider can halt the execution, create a snapshot, and send the snapshot to the respective resource consumer or another provider. Moreover, the provider can continue the execution right after the snapshot is created. The memory footprint of a snapshot is determined by the complete size of all elements which are necessary to continue the execution elsewhere. Further, each snapshot contains the computational effort that was necessary to achieve its state. This effort is measured in virtual machine instructions to make the effort comparable among different resource providers.

[11][61] is joint work with J. Edinger and C. Becker
[12][164] is joint work with J. Edinger, M. Breitbach, and C. Becker

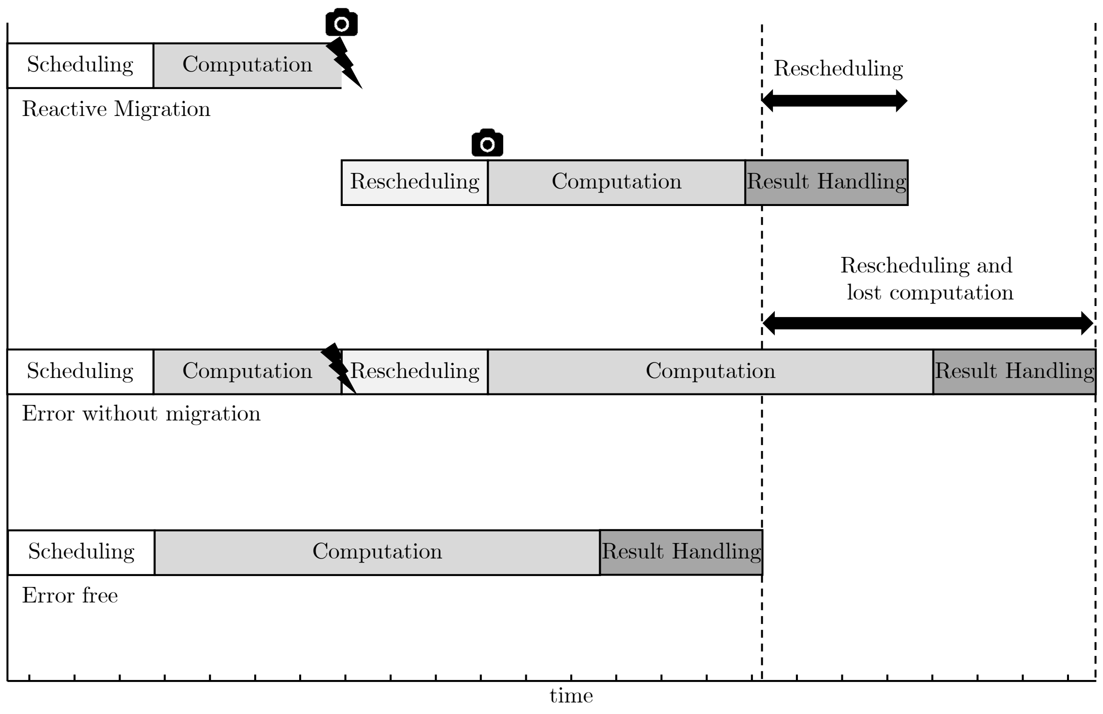


Figure 5.17.: The reactive task migration approach with an explicit leave. An error without any migration generates an additional delay of the rescheduling time (RS) plus the time for the lost computation (C'). When reactive migration is used, this time is reduced to only the rescheduling time (RS).

In traditional distributed computing environments, task executions can be aborted by the user or by a system error. Mobile devices are more dynamic in terms of battery status, network status, temperature, and location, which makes task abortion more likely. This leads to a higher risk of task abortions and must be handled differently. In case of a low battery, a decreasing network quality, or a rising device temperature, the middleware can explicitly stop the execution. However, a spontaneous location change or network disconnection would cut off the device completely and the execution progress is lost.

The task migration algorithm ensures no loss of execution for explicit system leaves. For implicit system leaves, the approach offers a quick start snapshot algorithm, which backups the computational progress in changing intervals. Therefore, the approach consists of two algorithms: *reactive* and *proactive migration.*

Reactive Migration

Figure 5.17 compares the execution delay of an unhandled error with the reduced delay of the reactive task migration algorithm. This illustration holds for explicit leaves or a task abortion, where executions are stopped gracefully. After stopping the virtual machine, a snapshot is created, and transferred to the resource consumer. The snapshot contains the complete execution progress done so far. Therefore, the resource consumer is able to reschedule the task without any loss of computation. The rescheduling is done analogously to the initial scheduling process.

Reactive migration has three effects on the overall computation time T from Equation 5.2. First, the fault detection time D_i is zero, since the middleware starts acting before a fault occurs. Second, the intermediate computation time C_i' is not lost. After the task is rescheduled, the execution starts exactly where the last TVM stopped. Therefore, the sum for all intermediate computation is equal to the computing effort with no errors occurring. Assuming that all devices are homogeneous in terms of performance and Tasklet executions are not paused, $C_{nofault} = C + \sum_{i=0}^{n}(C_i')$ holds, independent from n. Third, the RS_i includes the transfer of the snapshot, which is influenced by the size of its footprint.

Proactive Migration

Figure 5.18 compares the execution delay of an unhandled error with the reduced delay of the proactive task migration algorithm. In this case, an implicit system leave occurred, where mobile devices are disconnected from the network or shut down by the user spontaneously. Thus, the system cannot trigger the reactive migration. The intermediate state of a task is lost and the computation has to be re-initiated entirely. As a solution, proactive migration is introduced. It creates and sends snapshots to the consumer continuously. On the consumer side, only the latest snapshot of a task is stored. The middleware uses heartbeats to detect faults on the provider side. In case of missing heartbeats, the consumer re-initiates the task with the latest snapshot.

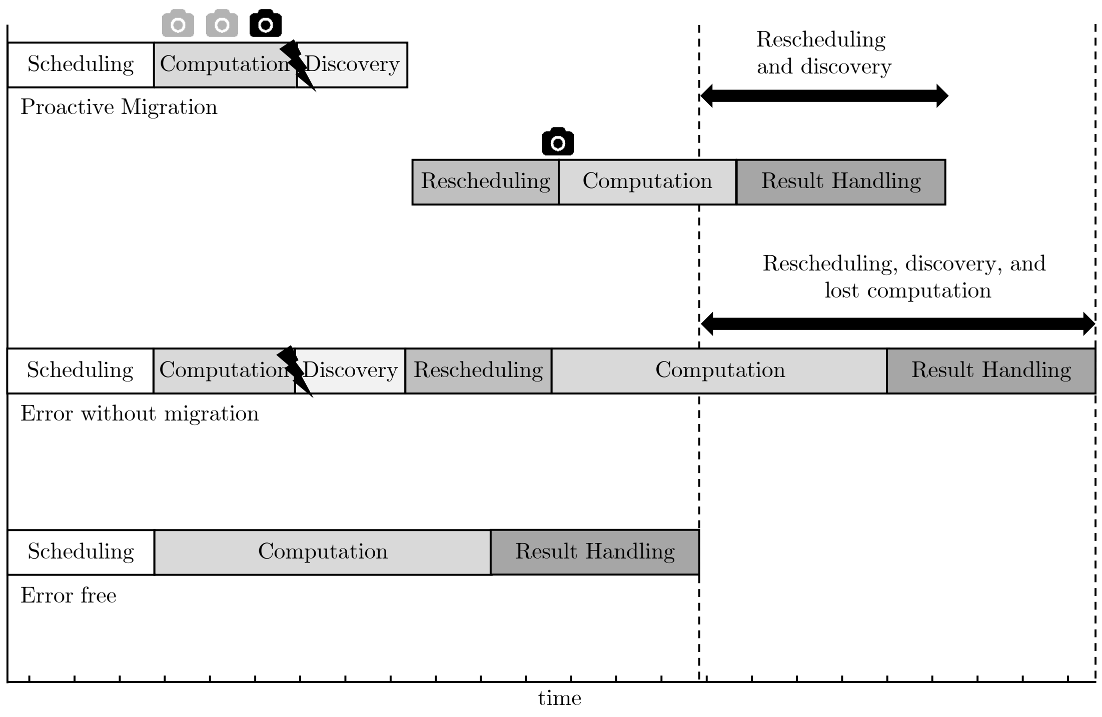


Figure 5.18.: The proactive task migration approach with an implicit leave. An error without any migration generates an additional delay of the discovery delay (D), the rescheduling time (RS), plus the time for the lost computation (C'). When proactive migration is used, this time is reduced to only the rescheduling time (RS) and the discovery delay (D).

In terms of the overall computation time T, proactive migration can be adjusted through the interval length between snapshots. There are two options: First, the application programmer can set a fixed amount of time in between two snapshots. Second, the quick start algorithm is developed that determines a snapshot sending interval depending on the current computational effort and the snapshot memory footprint size. At the beginning of a task execution, the interval is short and increases over time. The reason for that lies in the heterogeneity of tasks and relates to the execution effort of tasks. For short and highly responsive tasks, it is important to backup the early progress. For long running tasks, the snapshot interval can increase fast, since the potential relative loss is smaller.

Beside of the computational effort, the size of the footprint, the available bandwidth of the resource consumer, and the relative performance of the provider are important factors to determine the sending interval. This thesis only considers the footprint size. Figure 5.19 shows two examples of asymptotic wait functions that

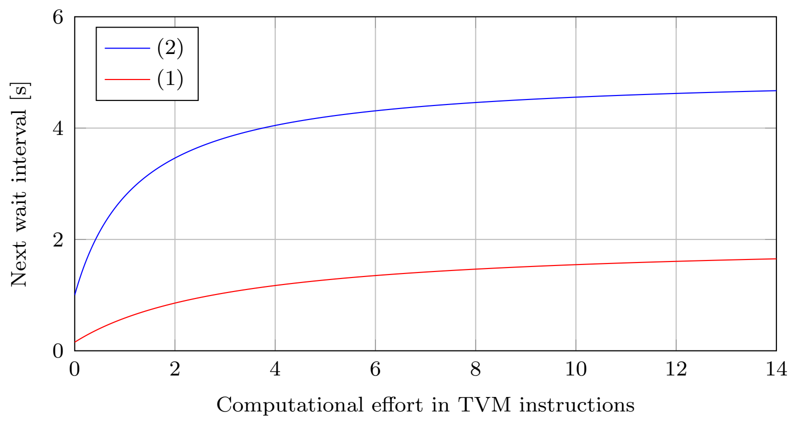


Figure 5.19.: Two different graphs to determine the snapshot frequency. The first one (1) is for short tasks with a small footprint, the second one (2) for long running tasks with a big footprint. According to Equation 5.6, the parameters Γ and ν are defined by the task properties. Based on these parameters, the function is determined by means of which the snapshot interval is computed.

are used to determine the corresponding snapshot interval. Equation 5.6 presents the general formula for the quick start approach. The parameter t determines the current computational effort of the task. The aforementioned factors influence the values Γ and ν. Γ determines the values that the wait function converges to and ν affects its slope. A high snapshot frequency decreases the potential loss of progress brought by a fault. However, this leads to overhead in terms of data transfer and virtual machine interruptions which are necessary to maintain consistency. Hence, the application programmer has to decide on that trade-off, which is application dependent.

$$wait(t) = \frac{\Gamma t}{\nu + t} \tag{5.6}$$

Since snapshots are created during the task execution time, the virtual machine is stopped for a short time period. This time is added to C_i' and C respectively. Implicit leaves are not predictable, which prevents proactive migration from recovering all states and data from the faulty resource provider. The computational effort that has been made between the last snapshot that arrived at the consumer and the fault that occurs on the provider determines the amount of lost computation. Therefore, $C_{nofault} \leq C + \sum_{i=0}^{n}(C_i')$ holds. The time for the detection of a fault D_i depends on the heartbeat rate and the sensitivity of re-initiation, which is the number of missing heartbeats until a task is re-initiated. In case of a high

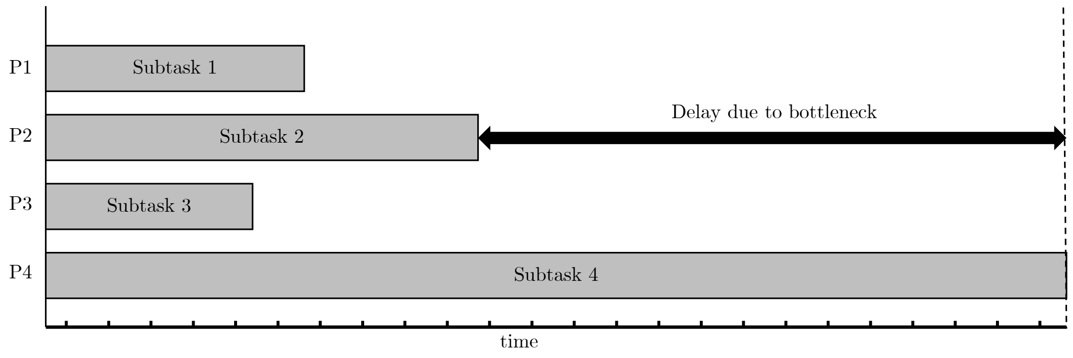


Figure 5.20.: The impact of execution bottlenecks. A task is split into four parts that are all required for the result assembly. The execution takes place on different resource providers (P1-P4). The slowest task determines the overall execution time of the task.

heartbeat frequency and sensitivity, D_i is short at the cost of heartbeat message overhead. Besides, false re-initiations of tasks can cause multiple concurrent executions of the same task on different providers. Once the middleware detects a fault, the rescheduling RS_i is analogous to reactive migration.

So far, fault tolerance mechanisms for computation placement in edge environments were presented, that cope with fluctuation and failures. Next, algorithms are proposed to avoid execution bottlenecks in heterogeneous environments.

5.4.5. Workload Partitioning for Bottleneck Avoidance

The second source for execution delays in edge environments are bottlenecks, which are shown in Figure 5.20. To parallelize computationally intensive applications, tasks are split in several subtasks and sent to different resource providers. After the computation, the consumer application requires all results of each subtask [13] to assemble the overall result of a task. This reduces the response times, but can also lead to large bottlenecks for two reasons: First, the computational capabilities of the employed resource providers are heterogeneous. Edge environments in particular are heterogeneous, since many different device types, like smartphones, laptops, and stationary PCs contribute their resources. Consequently, the slowest

[13] A computationally intensive part of an application is defined as a Tasklet bundle or task. Each Tasklet bundle consists of several Tasklets or subtasks. These Tasklets are all required to assemble a result for the consumer application. Thus, the latest Tasklet arriving at the consumer determines the runtime for its Tasklet bundle. Tasklets can further be split into several microtasks to reduce the risks of bottlenecks.

device will slow down the overall computation time. The second source of bottlenecks are irregular task structures, meaning, that the computational effort of a task is not linear to its parameter range. The last subtask of a computation can be much more complex compared to the first subtask. Further, both sources can occur at the same time, which aggravates the effect even more. In Figure 5.20, subtask 4 slows down the overall execution time and creates a bottleneck, which can be referable to provider P4, subtask 4, or both. This section presents a solution to cope with these bottlenecks based on [164][14].

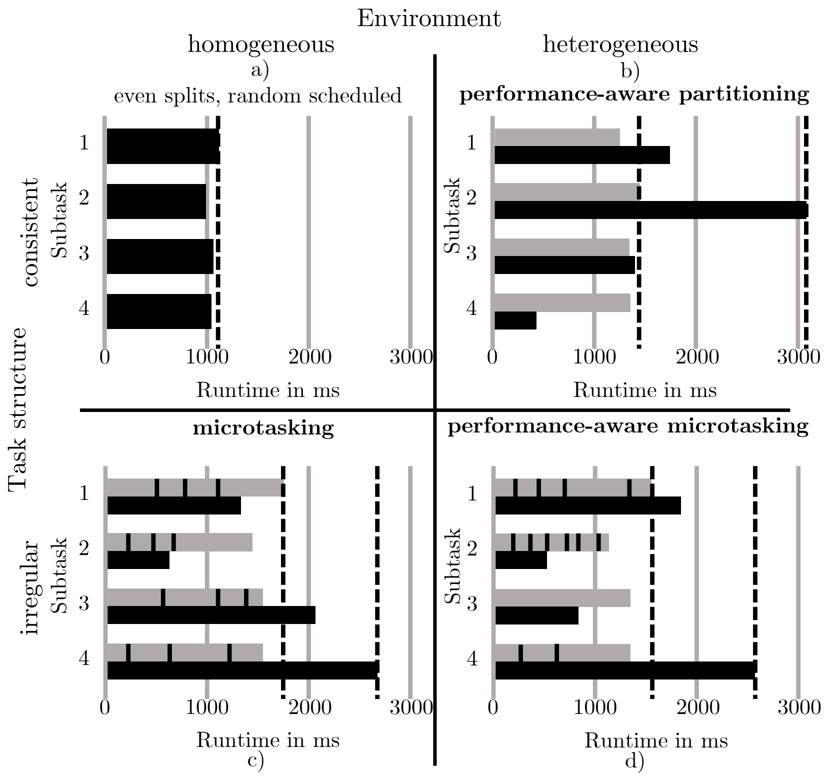


Figure 5.21.: Problems of environment heterogeneity and irregular task structures. In the optimal case (a), the computing environment is homogeneous and the structure of a task consistent. With heterogeneous devices in the environment (b), bottlenecks are likely. The same can happen when the structure of tasks is not consistent (c). When combining both (d), the impact is even stronger. As a solution, performance-aware partitioning and microtasking is introduced to cope with both issues.

Figure 5.21 pictures the problems of bottlenecks in more detail and outlines the solution design for environment heterogeneity and irregular task structures. Optimally, all devices in a computation environment and the offloaded tasks

[14][164] is joint work with J. Edinger, M. Breitbach, and C. Becker

are homogeneous, causing that the subtasks can be split evenly and scheduled randomly (see Figure 5.21a). Considering only the execution times, there are no bottlenecks in this case, since all devices process the subtasks in the same time. However, when the participating devices and tasks are not homogeneous, the chances for bottlenecks grow. The partitioning mechanism compensates for environment heterogeneity and task irregularity without any assumptions on runtime behavior or runtime estimation. According to Flynn's taxonomy [71], data level parallelism is therefore applied to the Tasklet system.

Automatic Data Level Parallelization

For the automatic parallelization task specific information is necessary that describes how a task can be split up dynamically during runtime. Therefore, the application programmer uses an API to specify information about the task data characteristics. The goal is to define the smallest possible granularity in which a task can be partitioned. Hence, the middleware can decide in how many subtasks a task is split up during runtime. This decision is made based on the current state of the computing environment, including number of resources, computational capabilities, utilization, and network connection.

Since the structure of task data can be different, the application programmer has three different categories to indicate its characteristics. These categories are *sets*, *ranges*, and *runs*. *Sets* encompass tasks that have numeric or symbolic input parameters without any sequence. One example for this parameterization is a k-means clustering implementation, where all potential cluster centers are parameters that are bundled into sets. With *ranges*, numeric intervals can be defined, by means of which a task can be parallelized. For that, image rendering algorithms are one example application. In this case, the range for the overall task starts with the first pixel and ends with the last pixel. Further, the smallest data granularity can be the computation of each pixel individually, depending on the algorithm. With this kind of task, the middleware has a high partitioning flexibility. The *runs* setting defines how often the same task is executed, which can be used for simulation applications. For example, a Monte Carlo-based method can use the *runs* setting to define how often a randomized experiment is executed to achieve a certain accuracy. These three categories are able to represent most computationally intensive parts of applications limited by the application model from Section 5.2.

Additionally, the approach includes so called *late parameter binding*. Therefore, it sends the data partitioning parameters along with the task to each resource provider. Before execution, the middleware replaces placeholders in the bytecode with the actual parameters, which are designated for the respective TVM instance. This decouples the compiling process from the scheduling as well as from the workload allocation. So far, the mechanisms only supports automatic data parallelism, without any consideration of heterogeneity or irregularity. Next, the improvements for automatic workload partitioning are presented.

Heterogeneous Environments

In contrast to a homogeneous environment, the devices in a heterogeneous environment can have the same accumulated computational performance, but not equally distributed among all of them. When tasks are split up into subtasks of the same size, the slowest device creates a bottleneck, since all subtask results are required to finish an execution. Especially with mobile devices, the performance range is large. This example is illustrated in Figure 5.21 b) by the black bars. Subtask two is scheduled on a slow machine and causes a bottleneck. The *performance-aware partitioning* approach is developed to cope with environment heterogeneity. The approach claims that it can achieve similar task completion times in heterogeneous environments. Therefore, each device runs benchmarks to determine its computational capability. Based on that, a performance index (CI) is calculated for each device individually and promoted in the system.

The resource query for a task contains the desired amount of computational performance. Depending on the current state of the environment, devices are selected to accumulate a reasonable performance index. The resource consumer middleware then starts the partitioning process. Based on the environment knowledge, it splits up the task in subtasks, by means of the three partitioning indicators mentioned above. These subtasks are not equally sized, but tailored to the allocated resource providers in terms of their performance indices. Consequently, a provider that can compute twice as much receives double the workload of another provider. In

Figure 5.21 b), the gray bars indicate the possible improvement introduced by performance-aware partitioning. As a result, the degree of performance distribution (or degree of heterogeneity) among a set of resource providers does not influence the Tasklet bundle runtime.

Irregular Tasks

In homogeneous environments, bottlenecks can also be evoked by task irregularity. Thus, it is not trivial to split them in unequal parts, since their structure is not consistent. For example, the computational effort to compute all prime numbers in an interval is not consistently distributed. The computation for the interval $100 - 200$ implies less computational effort than the interval $10,100 - 10,200$.

Therefore, a solution for heterogeneous tasks is introduced. Each subtask is split further into so called microtasks, which are then shuffled among the subtasks. Consequently, each subtask consists of non-sequential parts of the problem that all are executed in parallel on distinct providers. Hence, the risk of allocating a computationally heavy subtask to a single resource provider is spread, as shown in Figure 5.22. This means that a computationally heavy task T is split into several subtasks $T1$ - $T4$. These subtasks are split up further into microtasks ($T1_1$ - $T1_4$, $T2_1$ - $T2_4$, $T3_1$ - $T3_4$, and $T4_1$ - $T4_4$). These microtasks are shuffled among the four Tasklets, thus, decreasing the risk of receiving only heavy microtasks. This mechanism comes at the cost of allocation overhead and starting and stopping the virtual machine to execute non sequential partitions of a task. In Figure 5.21 c), the initial situation of task heterogeneity is shown by the black bars. Thus, the microtasking leads to smaller bottleneck risks and reduces the overall computation time, as shown by the gray bars in Figure 5.21 c). Figure 5.22 a) and b) visualize the microtasking approach in more depth analogously.

Heterogeneous Environments and Irregular Tasks

The most complex scenario combines a heterogeneous environment with irregular task structures. In this case, the potential for bottlenecks is the highest, since slow providers can receive computationally intensive subtasks. The solution is a combination of both algorithms, the so called *performance-aware microtasking.* To

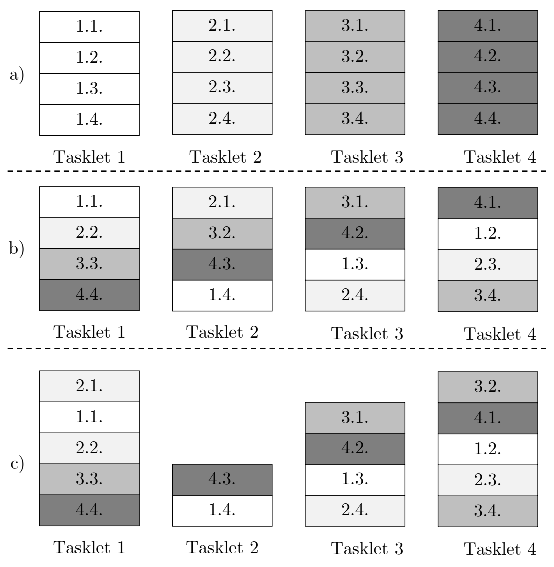


Figure 5.22.: A computationally intensive problem that is split into four Tasklets. The more complex a microtask is, the darker is its shade. a) shows the initial state, without any optimization. In this case, Tasklet 1 has the smallest and Tasklet 4 the highest computational effort. In b), the microtasking mechanism is applied and the microtasks are shuffled. This reduces the risk of including all complex microtasks in the same Tasklet. In c), performance-aware microtasking is applied and the number of microtasks per Tasklet is tailored to each resource provider individually.

realize that, the task is split into microtasks which are allocated to the providers considering their performance index. Hence, a more powerful device will receive more microtasks. For instance, a provider twice as fast will receive twice as many microtasks. In Figure 5.21 d), the third provider only executes one single microtask, compared to provider two, which executes seven microtasks. Figure 5.22 c) further illustrates performance-aware microtasking on Tasklet level.

5.5. Summary

This chapter presented a computation placement framework for edge environments, including an application model. It consists of the Tasklet middleware that supports the construction, orchestration, and execution of closed units of computation. To employ edge devices as elastic computation resources, the edge support layer was presented. This layer integrates different platforms, realizes ad-hoc as well as hybrid scheduling, and copes with fluctuation and heterogeneity at the edge. The next chapter introduces the implementation of the prototype.

6. Prototype Implementation

The previous chapter presented the design of the Tasklet system and the edge support layer. This chapter describes the implementation of the prototype, which is the foundation for the evaluation. The prototype implements the entire design of Chapter 5 and consists of several integrated artifacts. Due to the similarity of the design and the prototype implementation, the following chapter does not describe the entire architecture again but rather focuses on implementation-specific features of single components. The chapter is structured as follows: First, an overview over the general implementation details is given. In Section 6.2, the core Tasklet system is characterized. Third, the integration of various platforms is discussed. Here, the implementation details for various operating systems and hardware architectures, such as mobile devices, GPUs, and microcontrollers are given. Fourth, in Section 6.4, the implementation of the edge support layer features are presented: hybrid scheduling, workload partitioning, and task migration.

6.1. Overview

The prototype consists of several integrated artifacts facilitating different features. The first feature is the Tasklet core system that realizes the fundamental functionality of the design. It consists of about 9,400 lines of code and is written in C11. The core system is platform-independent and portable to most computing architectures with few exceptions. Especially the integration of GPUs involves some changes in the core system code, which is realized by an OpenCL GTVM implementation. The integration of mobile devices is based on Android and reuses the core system code entirely within Android Native Development Kit. Additionally, there is a component built in standard Android, to facilitate the interaction with the consumer application. This source code encompasses 740 lines of Java 8 source code and further 900 lines for the implementation of the decentralized broker component. The footprint of the thin Tasklet library is 20 kilobytes and

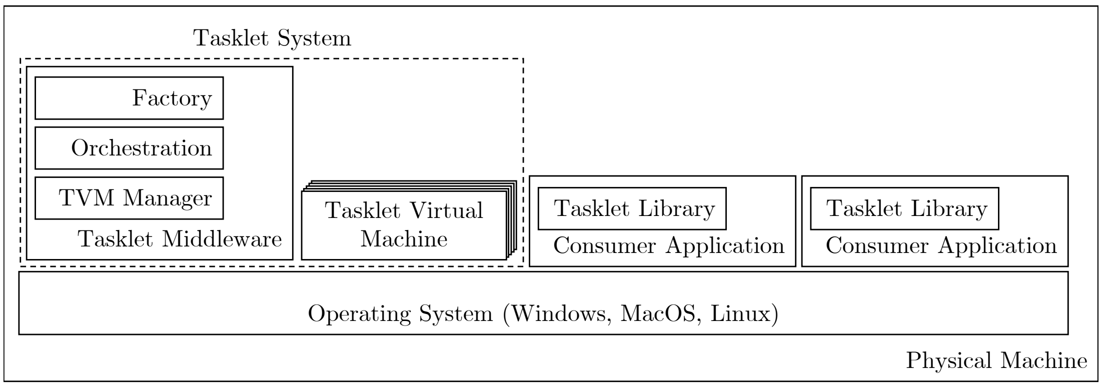


Figure 6.1.: Process model of the Tasklet system core. The Tasklet middleware consists of the factory, the orchestration, and the TVM manager. As separate processes, the TVMs run directly on top of the physical machine's operating system.

optimized for embedded devices. The implementation of a language-specific library supports the developer with an easy-to-use API for Tasklets. Therefore, libraries for Java, C#, and Android are implemented. The Java library consists of 26 classes, 2900 lines of Java 8 code. The C# library is similar to the Java library in terms of lines of code and functionality.

To extend the approach with elastic edge resources, a decentralized task allocation on nearby user-controlled devices is implemented, called the local broker. It runs on each participating device and extends the Mobile Tasklet implementation to collect context information, decide on resources for execution, and support the direct exchange of Tasklets. This approach is integrated into a hybrid scheduling mechanisms, combining cloud and edge resources. To increase the capabilities of edge devices as computing resources, their lack of reliability and homogeneity is taken into account. Therefore, the prototype includes the implementation of task migration and workload partitioning. Both mechanisms extend the standard Tasklet system components on several levels, ranging from the library's API to the TVM's parameter demarshalling.

6.2. Tasklet System Core

The Tasklet system core is divided into three parts: the middleware, the TVM, and the library. Figure 6.1 shows an overview of the implemented Tasklet system core. The middleware encompasses the factory, the orchestration, and the TVM manager. It is a standalone process. The factory is the main thread of the middleware and starts the orchestration as well as the TVM manager. In idle state, the middleware runs 10 threads simultaneously. After the TVM manager was started, it checks the user settings and starts TVMs accordingly. In the default setup, the TVM manager initializes one TVM per local CPU core. Each TVM runs in an individual process and communicates via the Tasklet protocol with the other components. The process isolation separates the TVMs from each other. Since unknown code is executed on the TVMs, this isolation ensures that heavy workloads, malicious code, or failures do not affect other TVM instances on the same physical machine. The third component is the Tasklet library, which is used by the consumer application. It facilitates the marshalling of Tasklet requests, the result handling, and the interaction with the Tasklet middleware via protocol messages. The broker is an additional component that runs a special version of the Tasklet middleware. All components written in C integrate the so called *Tasklet Environment Library*, which encompasses all system wide functionalities, including the Tasklet protocol, wrapper functions for portability support, and abstractions for Tasklet list implementations.

6.2.1. Java Library

This section presents the Tasklet Java library in detail. The C# library has similar functionalities and structure with some language-dependent exceptions. The library together with the Tasklet factory connect the consumer application to the Tasklet system. The library marshalls the plain Tasklet request and manages the connection with the Tasklet middleware. As seen in Figure 6.2 a), the library API provides an easy-to-use abstraction.

Figure 6.2 further shows the use of the Java Tasklet library and the corresponding C-- source code. In the first step, the Java variables are initialized and a Tasklet is created based on an existing source code file. This file is shown in Figure 6.2 b). Second, the variables are added to the Tasklet as parameters. The library supports a type-safe API that also considers the variable names. In case the execution requires data, the developer attaches it analogously to the parameters.

```
  public static void main(String[] args)
  {
1    int lower = 100, upper = 1000;
     Tasklet t = new
                Tasklet("primes.cmm");
  ----------------------------------------
2    t.addInt("lowerBound", lower);
     t.addInt("upperBound", upper);
  ----------------------------------------
3    t.setQoCReliable(GUARANTEED);
     t.setQoCSpeed();
  ----------------------------------------
4    t.start();
     int[] primes = t.waitForResult();

  }
```

a)

```
1 int lowerBound, upperBound, result;
  ----------------------------------------
2 procedure int checkprime (int a){…}
  ----------------------------------------
  >> upperBound;
3 >> lowerBound;
  ----------------------------------------
  while(lowerBound<upperBound){
      result := checkprime(lowerBound);
      if (result # 0){
4         <<result;
      }
       lowerBound ++;
  }
```

b)

Figure 6.2.: Integration of Tasklets into a Java host application for prime number computation. In a) the code is presented to (1) create a Tasket, (2) add parameters, (3) set QoC goals, and (4) start the Tasklet and receive its results. b) shows the respective C-- code file that consists of (1) variable declarations, (2) procedure declarations, (3) definition of variables with values from the host language, and (4) the main statement.

In the third step, the QoC goals for the Tasklet are set. Finally, the Tasklet is started and the results are retrieved. The last function call will block the program execution until the result has arrived entirely.

After a Tasklet is started, the library checks the validity of parameters and QoC goals. Next, it marshalls the entire Tasklet information and generates a Tasklet protocol message (presented in Section 6.2.3) with the plain Tasklet as payload. It forwards this message to the factory and waits for results. The library also supports a so called *Tasklet bundle*, which represents multiple Tasklets. The Tasklets in a bundle have the same source code and data, but can be parameterized individually. Based on that, the developer can split up tasks manually. Further, the Tasklet bundle is used by the partitioning mechanisms. The Tasklet bundle implements a further result handling method, called *waitForAllResults*. This method blocks until all results of a bundle are received. Tasklets have a timeout that is important for unreliable Tasklet executions. If the application is blocked and waits for results that will never arrive, the applications are in a deadlock. The timeout prevents this situation. After the results are received completely, the library unmarshalls the results and forwards them to the application.

To further improve the usability of Tasklets, an Eclipse IDE plug-in is developed. It offers a Tasklet language editor, a compiler integration, and a development support toolbar. The editor highlights keywords and functions of the Tasklet language and the compiler integration eases debugging of Tasklet code. The toolbar links the Tasklet language with the host language and gives an overview of variables that are transferred between the two languages for remote executions.

6.2.2. Factory

The factory compiles and assembles a Tasklet based on the plain Tasklet request from the consumer application. Each Tasklet middleware has one factory, which can be instantiated multiple times. In case of several consumer applications running on one physical machine simultaneously, the factory initiates one instance per application. Each application information is stored for unique identification to allocate incoming Tasklet results. Further, two distinct TCP connections are used for the transmission of plain Tasklet requests and Tasklet results.

The plain Tasklet request consists of the source code, the QoC goals, the parameters, and data. The source code is written in the Tasklet language C--, which is implemented according to the design. In general, the language has a C-like syntax and offers a subset of C's functionality. It supports integer, float, char, and bool data types for constants, variables, arrays, and function returns. Further, procedures can be parametrized, called recursively, and have a void return value, as most programming languages. The productions of the Tasklet language in extended Backus-Naur form can be found in Appendix A. C-- comprises 12 standard functions for math and array operations. Beside of the standard operators, it offers the Tasklet input << and output >> operator, as well as an array copy $A- > B$ operator. The array copy operator makes a deep copy of the left array A and stores it to the array on the right side B. In case that B is smaller than A, additional memory is allocated for B. In case A fits in B, B is overwritten with the content of A for the length of A.

The factory runs the C-- compiler that translates source code into byte code. Based on the host language concept, Tasklet parameters and data can be passed from the host language to the Tasklet. The factory unmarshalls the parameters and data from the plain Tasklet request. The compiler starts the translation process, incorporates the parameters, and links the data to the byte code according to position of the Tasklet input operators << in the source code. While doing that, the compiler checks the data types and validates if the correct parameters were parsed. Constant values of primitive data types are directly inserted in the byte code. The same applies for variable parameters but, further, their positions are marked in the byte code. This speeds up the execution in the TVM, but also gives the opportunity to change the parameters later on. Larger data is written in the constant pool of a Tasklet and inserted as references in the byte code. The constant pool is a well known construct for building bytecode interpreters [147, p. 242] and stores all elements that do not fit into a 4-byte integer.

Regardless of the type of parameter or data that is transferred from the host language into the Tasklet, the compiler marks the positions and caches the bytecode in the factory. The reason for that is the reparameterization of Tasklets, which is used to speed up the Tasklet assembly time. Most consumer applications start the same Tasklet with different parameters or data several times. After the first Tasklet request is sent from the library to the factory, the library eliminates the overhead of sending the source code repeatedly. Therefore, it sends a specific code reuse message, which only contains parameters and data. The factory directly assembles the Tasklet and uses a bytecode reparameterization mechanism. This mechanism retrieves the bytecode from the factory cache and places the new parameters and data accordingly. By applying this mechanism, data transfer and compiling overhead can both be reduced.

6.2.3. Orchestration

The orchestration implementation consists of the Tasklet protocol, the broker network, and the QoC mechanisms.

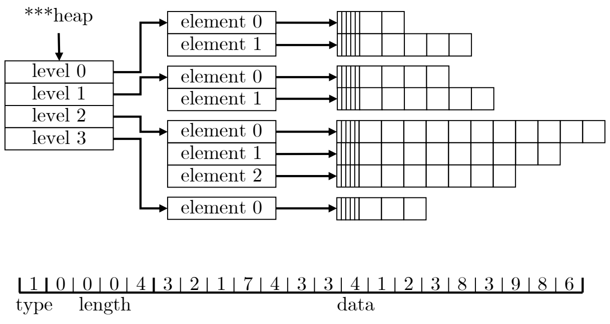
The Tasklet middleware uses the *Tasklet protocol*, which facilitates the internal and external communication. It encompasses 32 different message types, which can be subdivided into four message classes. *Interface messages* are for external communication with the consumer application in both directions. *Broker messages* realize the communication with brokers. Both, consumers and providers communicate with their broker for requesting and registering resources, respectively. *Tasklet messages* are messages that contain assembled Tasklets or their results. This type of message is used to transmit a Tasklet from the consumer to the provider orchestration, but also for sending the Tasklet to the TVM locally. Messages that contain a Tasklet snapshot for migration belong to this class as well. All other types of messages are *management messages*, which are used for example for heartbeats, status updates, and Tasklet cancellations. The Tasklet protocol data units consist of a header with a magic, a version number, the corresponding message type, and the message payload. Additionally, each message has an own header that specifies its payload. The Appendix B contains an overview over the protocol messages.

The prototype uses a single *broker* instance rather than a network of multiple brokers. The single broker instance was capable of serving around 200 providers in the evaluation [166]. To further scale up the system, multiple brokers can be used to balance the load of requests. The broker is started on a stable node in the network. Resource consumer and provider register at the broker and therefore need to know its IP address.

The Tasklet orchestration facilitates the *QoC mechanisms* such as reliability, multiple execution, and speed. The orchestration extracts the QoC settings of each Tasklet message that arrives and triggers the corresponding mechanisms. Depending on the QoC, this can lead to Tasklet duplication, monitoring of heartbeats, or a specific resource selection. The further implementation of QoC mechanisms is not in scope of this thesis. More information can be found in [166].

6.2.4. Tasklet Virtual Machine

The TVM is the runtime environment for Tasklets. All TVMs on a provider are managed by the TVM manager, which allocates Tasklet in round-robin fashion on idle TVMs. The TVM encompasses a stack-based bytecode interpreter and a runtime environment, which enables Tasklet executions similar to batch jobs. The TVM is single-threaded and does not support concurrent Tasklet executions. It is non-preemptive and therefore does not support context switching between Tasklets. The bytecode interpreter consists of a stack, a stack pointer, a program counter, a base address, a constant pool, and a heap memory for dynamic data types. For each procedure call, a new activation record is created on the stack.



a)

```
int [2] aGlobal;
Int [5] bGlobal;
procedure void c(){
          float[] aC;…}
procedure void b(int a, int b, int c){
          float[a] aB;
          int[b] bB;
          int[c] cB;
          c(3);…}
procedure int a(int a, int b){
          float[a] aA;
          int[b] bA;
          b(11,9,7);…}
<<a(4,6);
```

b)

Figure 6.3.: Structure of the heap space. a) shows the state of the heap, after function c was called. It further shows an example integer (type = 1) array with the length of 4. b) shows the respective code that creates the heap space from a).

The heap structure is analogous to these activation records, i.e., for each function call a new heap level is created. The heap is represented by a pointer that has three indirections. The first indirection determines the activation record level of the stored data item. A global array of a Tasklet would be created on the lowest level and is therefore visible on all higher levels. The second indirection defines the element within a scope, which then points to a byte array that stores the actual values. Each byte array begins with five bytes, describing the type and the length of the element. The heap structure facilitates the allocation of dynamic data types. With the array copy operator, arrays can be re-sized during Tasklet runtime. Figure 6.3 a) shows the state of the heap space with the respective .cmm code right after function *c* was called as well as the structure of a heap entry.

The workflow of the TVM is as follows: (i) After a TVM starts, it runs a benchmark, initializes sockets, variables, and registers at the TVMM locally. The benchmark determines the performance index of a resource to evaluate its relative computation capability in the system. (ii) It starts the main thread and a management thread, which is responsible for pausing, resuming, and terminating the TVM. (iii) The main thread waits for new incoming Tasklet executions. (iv) When receiving a Tasklet, the TVM is initialized with the byte code, the constant pool, and the execution parameters. (v) The byte code interpreter starts the Tasklet execution and adjusts the stack size, if necessary. (vi) After the execution finished, the TVM checks for execution error. If the execution was completed without errors, the

Tasklet result is send to the local orchestration. In case of errors or an incomplete execution, the orchestration if notified. (vii) The interpreter is reset and the TVM goes back to step (iii).

After the description of the Tasklet core system implementation, the realization of the platform support is presented next.

6.3. Platform Support

A major goal is to include common computing platforms to utilize computational resources in the environment. The Tasklet system core is written in C and, thus, originally not platform-independent. As a solution, the platform-dependent system calls in the Tasklet prototype are encapsulated in the so called wrapper environment. This includes the socket communication, the thread management, the mutex operations, the file system accesses, and other calls that return the current system time or the number of CPU cores. The wrapper environment knows on which operating system it is executed and uses the corresponding library. For example, the libraries that Windows needs for managing threads and using sockets are *process.h* and *winsock2.h*, respectively. For Unix-based systems, the *pthreads.h* and *sys/socket.h* libraries are used. Based on that, the prototype runs on Windows, Linux, and MacOS systems. Other computing platforms like mobile devices, GPUs, and microcontrollers differ from standard PC architectures in such a way that they need a more tailored implementation.

Next, the integration details of these platforms are elaborated.

6.3.1. Smartphones and Tablets: Mobile Tasklets

The following section is based on [163][1]. The Mobile Tasklets prototype is written for the Android operating system. The properties of Android had influences on the implementation of the approach. Android activities are responsible for user interactions but not suitable for computational load. In contrast, Android services are specialized on computational intensive background tasks. The Android Native Development Kit allows the developer to run *C* code and to access the

[1] [163] is joint work with J. Edinger, T. Borlinghaus, J. M. Paluska, and C. Becker

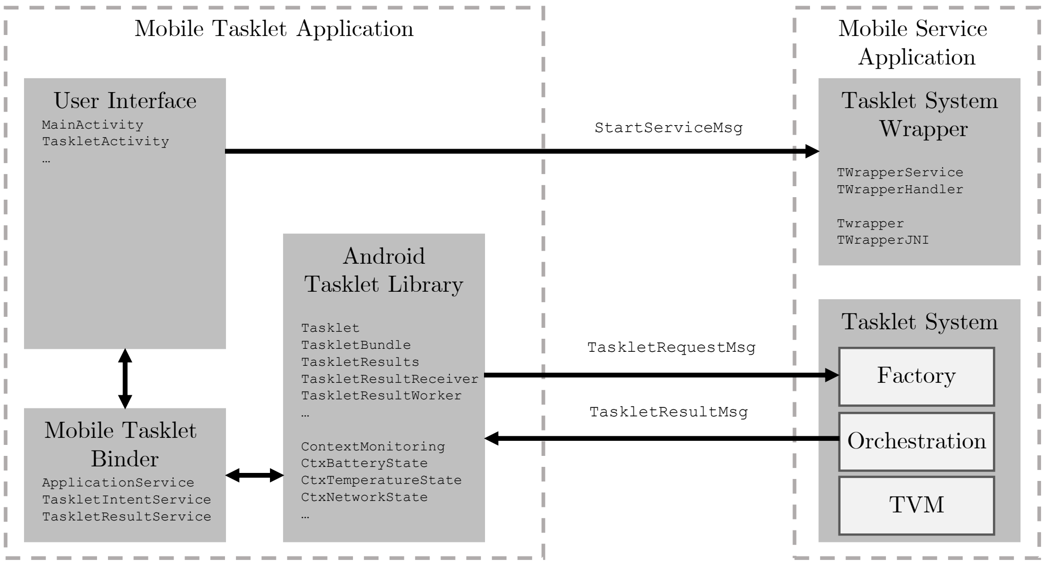


Figure 6.4.: Mobile Tasklet implementation. It encompasses an Android specific library and a Mobile Tasklet Binder that together link the consumer application with the Tasklet system. On the service side, the Tasklet system core is running within the Android NDK environment.

physical device components. Consequently, the design has two components that are represented as two distinct applications running on an Android device for the integration. First, the *Mobile Tasklet Application* handles the application interaction, the creation of Tasklet requests, and the initialization of the Tasklet core. Second, the *Mobile Tasklet Service* is a background service that enables the interaction with the Tasklet system core. These components run natively in the kernel of the Android device. Figure 6.4 shows an overview of the Android implementation and the interaction of the components. The Mobile Tasklets architecture is fitted to Android devices, however, with some adaptations also other platforms can be integrated. Since iOS also runs native C code, the transition to Apple devices would also be possible.

Next, the application and the service are explained in more detail.

Mobile Tasklet Application

The Mobile Tasklet Application handles the interaction with the user, assembles and dispatches Tasklets, and holds a component to monitor the device's context. The Android Tasklet library provides an API and can be included in any Android application. Tasklets can easily be created, started, and their results can be retrieved. The Android Device Monitor observes battery, network, and temperature states. Thus, it enables to react on certain events to retain quality of service levels. The Tasklet Binder connects the user interface to the Tasklet library. The Mobile Tasklet application only handles light computational tasks to guarantee a smooth user interface.

The *Mobile Tasklet Binder* interacts with the user application to exchange Tasklet requests and Tasklet results. It abstracts the underlying file system by converting Android file accesses into Tasklet file formats. It further decouples the user interface from operations like data conversion and Tasklet request generation.

The *Android Tasklet Library* interacts with the user application. It is not doing heavy computational work other than the assembling and dispatching of the plain Tasklet requests. Further, it is accessed through the Mobile Tasklet binder, to ensure encapsulation and isolation from the user interface. Thus, the user interface does not need any information about the Tasklet library. The Android Tasklet library is similar to the standard Tasklet library, but enables an easy-to-use Android app integration. To implement custom Tasklet applications, the developer has to extend the *TaskletActivity*-class. It encapsulates the entire logic for binding the app to the Mobile Tasklet binder and the Tasklet system wrapper. Further, it offers an easy-to-use API for all Tasklet related functionalities.

The *Android Device Monitor* acquires the device's context and triggers notifications. It is a subcomponent of the Android Tasklet library. It monitors the network connection, the battery usage, the temperature, and the workload of the device. For example, it pushes notifications about network or power mode changes. Depending on the event, the execution of Tasklets can be aborted, paused, or continued.

Mobile Service Application

The Mobile Service Application is designated for long-running tasks. It does not receive direct user input, but coordinated messages from the mobile application. The Mobile Tasklet service is responsible for initializing and maintaining the Tasklet system core. Further, it identifies times when devices are idle and sets the Tasklet system in standby mode to safe energy.

The *Tasklet System Wrapper* connects the *Tasklet system core* and the user application. All applications that run on a device and use Tasklets first send an initiation request via *Java native interface* (JNI) to the wrapper. The wrapper keeps track of these applications and enables a fair resource allocation among them. As long as there are applications using the Tasklet system, the wrapper keeps the Tasklet system running or in standby.

The *Tasklet System Core* contains the entire runtime environment of the Tasklet system. This includes the mobile versions of the orchestration, the factory, and the TVM. The components are similar to the non-mobile version of the Tasklet core.

Next, the implementation of the integration of thin devices is presented.

6.3.2. Thin Clients: The Tasklet Gateway

For the integration of thin devices, such as embedded devices, microcontrollers, and IoT sensors, two independent components were implemented: the thin Tasklet library and the Tasklet gateway. The Tasklet gateway is written in Java 8 and consists of $1,000$ lines of code. It is built to run on a stable host, which can be an edge or cloud resource. The gateway mimics a standard consumer application from the perspective of the Tasklet system. For thin devices, however, it offers a stable end point and takes over computational load for Tasklet creation and scheduling. The thin Tasklet library is written in C11, consists of $1,100$ lines of code, and has a memory footprint of 20 kilobytes. It runs on all thin devices that are C-compatible and connects the thin device with a Tasklet gateway. Hence, the thin device does not need to run the Tasklet middleware and the computational load is handled by the Tasklet gateways. The thin Tasklet library offers a simple API for the developer that provides functions to connect to the gateway, send a plain Tasklet request, and retrieve the results. The abstraction, however, is on a lower level, compared to the standard Tasklet library. Marshalling and unmarshalling of data is not entirely done by the library.

To identify an application after it changes its network connection, the gateway maintains unique identifiers for all its clients, decoupling the application from the device's IP address. Further, the gateway buffers results in case it cannot contact the initiating applications immediately. This is important, since thin devices may be connected via an unreliable network link. The thin Tasklet library sends updates about its connectivity status and IP address changes in order for the gateway to forward the results to the correct device. To reduce the network traffic for Tasklet execution requests, the gateway caches entire execution requests. In case the same Tasklet is started again, the initiation can be triggered with minimal message effort.

While the gateway integrates the thinnest devices of the computing landscape, the GTVM exploits GPUs as computational resources, as presented in the next section.

6.3.3. GPU-Acceleration: The GTVM

The section is based on [161][2]. For the integration of GPUs, the two components presented in the design were implemented. The *GTVM manager* is a standard operating system process that runs on the CPU. It administrates the second component, the GTVM, running on the graphics card of the resource provider. The GTVM is written in OpenCL to create a portable solution, which can be deployed on all kinds of GPUs. The alternative for this approach would have been to implement a native CUDA program for NVIDIA GPUs, resulting in a better performance at cost of portability.

Figure 6.5 shows the overall architecture of the implementation of the GPU integration. It considers the special characteristics of GPUs as stated in the design. Each Tasklet provider runs a GTVM manager component, which identifies, benchmarks, and manages all GPUs of a computer system. For each GPU, a proxy is started. On each execution instance, a kernel runs a Tasklet thread, which represents a single Tasklet execution and is comparable to a standard TVM. The proxy compiles a kernel based on a Tasklet that is then transferred to the GPU for execution. The GTVM uses shared memory to provide common resources, like the program text, to all concurrent execution instances. In Figure 6.5, the system has two GPUs, which is common for most Intel computer systems, since they have an integrated GPU in their CPU. Therefore, the larger GPU can be dedicated as a co-processor for GPU-accelerated Tasklet execution. The communication between the GPU and the CPU is done via status flags. Each Tasklet kernel has

[2][161] is joint work with J. Edinger, and C. Becker

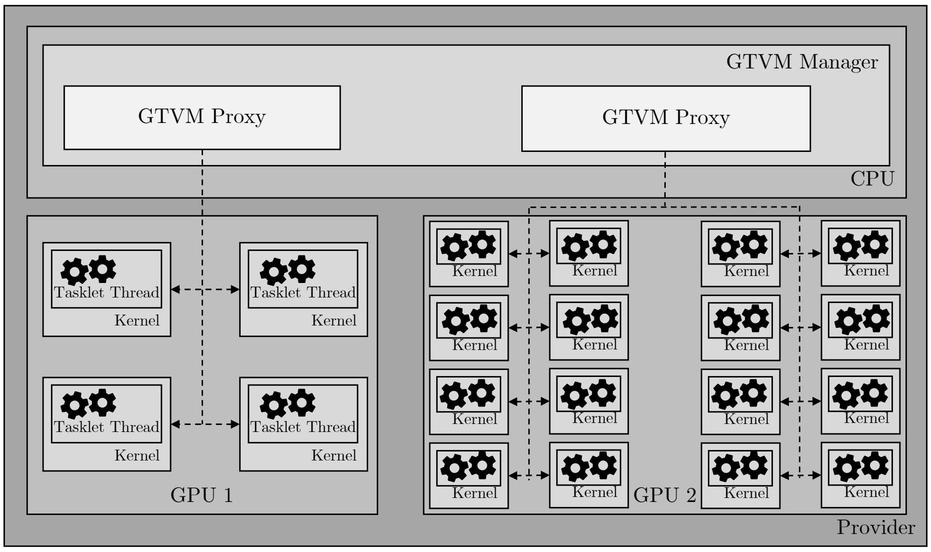


Figure 6.5.: Overall architecture of the *GPU-based Tasklet Virtual Machine* (GTVM). The GTVM manager and GTVM proxy run on the host side of the system - on the CPU. Depending on the number of streaming multiprocessors and processing cores of the GPU, the GTVM proxy compiles and starts several kernels on the GPU. Each kernel runs a Tasklet execution in form of a Tasklet thread.

a flag, which represents the current state of the thread. In case the thread is terminated, the corresponding flag indicates that the results can be retrieved or an error has happened. The following section goes into detail about the execution process and the adjustments to the Tasklet runtime environment.

Execution Process

To execute Tasklets on the GPU, the following process is necessary. The GTVM manager and proxy are the so called host side. They are processes executed on the CPU. The actual execution kernel runs on the GPU, the so called device side. The host side takes over all pre- and post-processing as well as the execution monitoring. This includes kernel compilation, memory allocation, data management, kernel invocation, and result handling. During the Tasklet execution, the host side monitors the GPU based on the status flags. Besides, the GTVM manager establishes the connection to the Tasklet middleware.

After receiving a Tasklet request, which is designated for the GTVM, the respective proxy initializes the setup. Then, the memory allocation is done depending on the Tasklet size. The host side compiles the OpenCL kernel, which consists of the interpreter and the bytecode. The GTVM is reset to clear all prior execution states. Next, the GTVM proxy copies the kernel to the GPU memory, including the bytecode, the parameters, and the data. The kernels are now invoked and the execution state is observed. In case of runtime errors, the execution terminates and communicates the error via the status flags.

After the execution, the results are retrieved. Three different result retrieval strategies are part of the design: *WaitForAll*, *JustInTime*, and *Overflow*. *WaitForAll* implies that the result is retrieved as soon as all Tasklet threads have terminated. This strategy maximizes the memory bandwidth utilization between the host and the device side. However, results may wait some time on the device side before they are forwarded. The *JustInTime* strategy handles the results of kernels individually and retrieves them as soon as they are completed. A major advantage of this strategy is that, right after an execution, the occupied memory space is cleared. Therefore, new Tasklet threads can be invoked. Moreover, time-critical results can be forwarded directly. However, since every result is copied individually, this reduces the effective usage of the memory bandwidth between the host and the device side. The *Overflow* strategy retrieves the results immediately when the allocated result memory of a kernel is full. In scenarios where only a small amount of memory can be allocated for the results or the result size is large, this strategy can be beneficial.

Parallel Architecture Support

Parallel computing architectures like GPUs differ strongly from a standard x86 architecture in terms of memory, execution modes, and scheduling. Therefore, some parts of the Tasklet system need to be adapted to integrate GPUs without changes in the Tasklet API. The computation abstraction is the same, meaning, that the same bytecode runs on GTVMs and TVMs. The adaptation of the Tasklet system mostly takes place in the TVM and in the orchestration, which interact directly with the TVM. The core of the TVM is translated into the OpenCL kernel, which is instantiated on the GPU several times to achieve parallelism. Minor adjustments affected programming language specifics, like function arguments, address spaces, arrays, and random number generation. However, the main challenges of the integration are memory management, multiprogramming, and kernel caching, which are described in particular.

Memory Management: The memory management in the standard TVM uses the dynamic memory model of C11. The system design is based on OpenCL version 1.2., which is necessary to support NVIDIA devices. However, this OpenCL version only offers static memory management, meaning, that the memory cannot be reallocated at runtime. The TVM has three different memory types: text, stack, and heap. The stack and heap sections grow during the execution of a Tasklet. Therefore, further mechanisms are necessary to allocate memory in the GTVM dynamically. The standard Tasklet system uses dynamic memory management in form of reallocation. For the GTVM, a memory management was implemented. It statically allocates memory before the execution, splits the memory in even parts for all concurrent Tasklet threads, and allocates the residual memory during runtime. In case of memory overflows, the number of running kernels is reduced.

Multiprogramming: In computer systems with only a single GPU, the operating system needs the GPU periodically to display the GUI. The GPU is scheduled non-preemptively, i.e., the executing process cannot be stopped by the operating system gracefully . In case a user program blocks the GPU for a certain amount of time, the operating system kills all GPU user processes to regain the control. Thus, the GTVM pauses and resumes Tasklet executions periodically, to allow the operating system in the intervening time to render the GUI. For this, the entire memory state of the OpenCL kernels needs to be stored. There are three types of memory on the GPU, which have different behaviors regarding their volatility. The private and local memory of a streaming multiprocessor is highly volatile compared to the global memory, which is not cleared in between process switching. Therefore, the state must be stored in the global GPU memory. Furthermore, the time intervals between two execution pauses can be adjusted. With short execution intervals, the GUI can run smoothly, however, this introduces a large overhead and slows down the GTVM performance. On the other hand, long intervals of execution are more effective in terms of Tasklet performance, but the probability of disturbing the user is high. The evaluation investigates different options for this parameter to achieve high performance without interfering the user.

Caching Strategies: The GPU architecture introduces caches, which can optimize the runtime by reducing memory latencies. With OpenCL, constant variables are automatically placed in the local memory of a streaming multiprocessor. This has rather a small benefit, since the amount of constants in the present system is small. The kernel has three caching options: *stack*, *code*, and *both*. Depending on the selected option the stack, the code, or both are transferred from the global to the local memory cache. This cache is cleared when the execution is paused due to the multiprogramming mechanisms. Therefore, to protect changes in the data, the content is flushed to the global memory. Compared to the bytecode, which is static, the stack is constantly changed during the execution and, hence, must be stored in the global memory every time. This introduces a certain overhead, which is investigated in the evaluation.

After the platform implementations the next section introduces the components that support elasticity and fault tolerance in edge environments.

6.4. Edge-centric Components

So far, the chapter presented a system implementation that facilitates code offloading from generic applications to heterogeneous platforms for computation. Next, analogously to the design, the system is specialized towards edge-based computation placement. The characteristics of edge computing are heterogeneity, fluctuation, and unreliability [164]. In this section, the implementation of decentralized edge scheduling, task migration, and workload partitioning is presented. Together, these system components allow the Tasklet system to make elastic use of edge resources while taking their characteristics into account.

6.4.1. Decentralized Scheduling in the Edge

This section is based on [165][3]. The local broker is a standalone Android application, which is connected to the Mobile Tasklet application via JNI. It has three components: the network handler, the context engine, and the Tasklet connector. It runs in parallel to the standard Mobile Tasklet application, explained in

[3] [165] is joint work with J. Edinger, J. Eckrich, M. Breitbach, and C. Becker

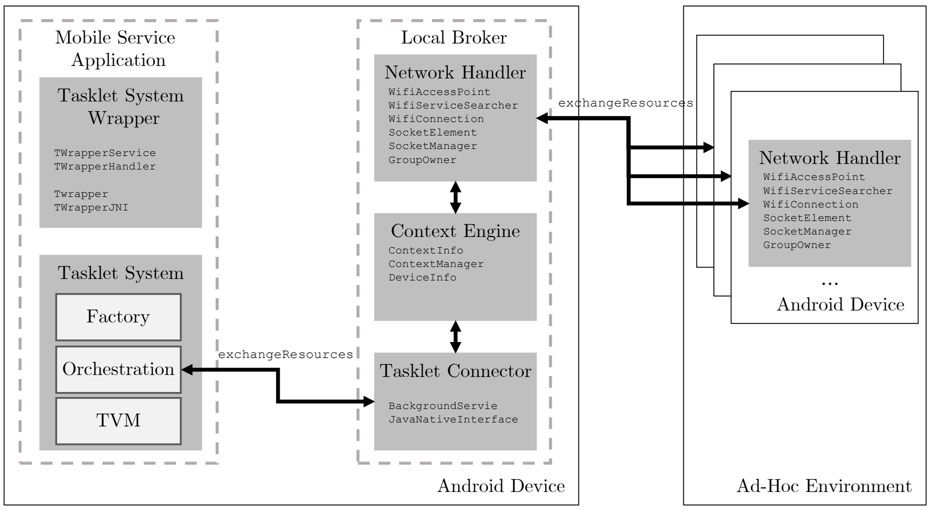


Figure 6.6.: The local broker has three components: the network handler, the context engine, and the Tasklet connector. On the left side, the Mobile Tasklets service is shown that is connected to the local broker via JNI. On the right side, an ad-hoc environment is shown. The local broker instance is connected with all devices in the environment via Wifi Direct.

Section 6.3.1, which was extended with the hybrid scheduling mechanisms. This mechanism checks the available network connections and decides to schedule on remote or nearby resources. This decision takes place when a Tasklet request was submitted and assembled by the factory. Subsequently, the orchestration requests resource providers at the broker. At this point, the hybrid scheduling decides whether to request the remote broker system or the local broker system. For the remote request, the Tasklet orchestration uses the standard mechanisms via a TCP connection to the broker. The nearby resources are requested from the local broker, which runs in the mobile device's Java environment. This request is sent by the Tasklet orchestration from the native Android environment via JNI. The local broker network is connected via Wi-Fi direct. To realize the use of remote and nearby scheduling in parallel, two options exist: A mobile device can use the 3G mobile network for remote scheduling, while connected to a nearby edge group via Wi-Fi. For the second option, tethering is used within the ad-hoc Wi-Fi environment to connect all devices to the Internet as well. However, this approach has the major drawback that all devices communicate through the access point with remote resources. Figure 6.6 shows the implemented prototype of the local broker.

Next, this section explains further detail about the context engine and the network handler. The *Tasklet connector* is not further examined, since its main functionality is connecting the Mobile Tasklets system with the local broker.

The Context Engine

The *context engine* gathers information about the local device. A major benefit of implementing the local broker in the Java environment and not in the native kernel of the device is the easy accessibility of all kinds of context information via the Android API. It offers all sorts of classes to easily read information about the network, temperature, battery, and other device contexts via Android intents. By using Android's *WifiManager* class, the Wi-Fi signal strength is read and automatically classified into a level between zero and five. Information of the battery state is also classified into the categories unknown, charging, discharging, not charging, and full. Additionally, the battery level is requested via an Android intent. The temperature of the device must be read from a file, which is located in a system directory in the internal memory. Lastly, the additional information about the device, like number of processors, current CPU utilization, or the hardware model can be read via the *DeviceInfo* class. This context information is gathered by the local broker and used for the allocation of Tasklets. The current prototype applies the CPU temperature, the signal strength, and the battery context.

The Network Handler

The *network handler* component handles all network- and connection-related tasks. The approach uses a network underlay, which is built with Android WP2P technology, and a network overlay that is used for Tasklet allocations. Android WP2P complies to the Wi-Fi Direct specification, which enables device-to-device connectivity based on the IEEE 802.11 infrastructure mode. Therefore, a device can act as a standard client or as a so called *group owner (GO)*, which takes over the tasks of an access point (AP). A special feature of Wi-Fi Direct is that these roles are negotiated dynamically during the network initiation. After two devices created a P2P group, other device can join as clients. The GO acts as a gateway to the Internet.

The standard way of connecting devices based on the Android WP2P involves user interaction for confirmation. Since the local broker runs autonomously, this is not applicable. As a solution, a well-known workaround is used. It makes use of the idea of initially creating multiple GOs and deleting the redundant ones after the network has been initialized. It consists of seven steps[4]: (i) Each local broker instance that does not belong to a group creates a Wi-Fi AP. (ii) The local broker extracts the password from the created network. (iii) It advertises the SSID and password by using a broadcast in the nearby Wi-Fi. (iv) Each local broker switches to a service discovery mode periodically to receive the broadcast of others. (v) When it receives a broadcasts from another AP, it extracts the SSID as well as the password. (vi) The local broker then connects to the other AP. (vii) Finally it deletes its own advertising service as well as the AP. This identifies one of the brokers as the GO. After the network has been initiated between two devices, others may join. In case the GO leaves the system, all participants start the mechanism again and determine a new GO.

Next, the Tasklet migration and workload partitioning implementations are presented next. Both section are based on [164][5].

6.4.2. Tasklet Migration

The implementation of the migration mechanisms is mostly realized in the Java library, the orchestration, and the TVM. In the library, a new API method was added to activate and parameterize the migration algorithms:

```
TaskletBundle.setMigration(boolean reactiveMigration,
                boolean proactiveMigration, int timeInterval);
```

The first two boolean parameters enable the reactive and proactive migration approach respectively. In case the proactive migration is enabled, the third parameter can be used to set a fixed time interval in between two snapshots. To use the context-aware determination of the snapshot interval, the developer has to set the value of the last variable to zero. The migration information is appended to the Tasklet closure and can be obtained anytime.

[4]Based on solution of Dr. Jukka Silvennoinen: https://github.com/DrJukka
[5][164] is joint work with J. Edinger, M. Breitbach, and C. Becker

The creation of snapshots is essential for both migration algorithms. A snapshot represents the entire state of the TVM including the program counter, the stack, the heap, and all intermediate results, which were created by the Tasklet execution so far. To create a snapshot, the TVM is interrupted and paused until all states are copied. An interrupt thread runs in parallel with the execution thread and has all information about the desired snapshot interval settings. After an interrupt occurred, the thread computes the time to wait until the next interrupt. This is based on the memory footprint of the snapshots, the computational effort, and the bandwidth between the provider and the consumer. To interrupt the TVM, the interrupt thread sets a vector to a certain value. After each instruction, this vector is checked by the TVM. If it is not zero, the TVM interrupt handler is scheduled. After that, the TVM may resume the Tasklet execution.

In case the reactive migration is enabled and the provider stops the Tasklet execution gracefully, a snapshot is created. Afterwards, the snapshot is sent to the orchestration of the resource consumer. The provider then stops the execution of the Tasklet. The orchestration receives the snapshot and requests a new provider to continue the execution. Next, the Tasklet snapshot is transferred to the new provider, which continues the execution right at the point, where the last provider has stopped. To do that, the program counter, stack, heap, as well as intermediate results are loaded into the TVM and the execution is started.

In case the proactive migration is activated, the TVM sends a snapshot to the orchestration of the respective consumer and continues the execution. After receiving the snapshot, the orchestration looks up the corresponding Tasklet entry and stores the snapshot. Only the most recent snapshot is stored and old versions are deleted immediately. When the orchestration detects that a provider left the system without notice via the standard heartbeat mechanisms, the orchestration looks up if a snapshot is stored. If that is the case, the snapshot is used for the re-initiation of the Tasklet execution.

Next, the implementation of the workload partitioning algorithms is presented.

```
public static void main(String[] args) {
  int lower = 100, upper = 1000;

  Tasklet tB = new Tasklet("primes.cmm");

  tB.setPartitioning(lower, upper, 1, true, true);

  tB.start();

  int[] primes = tB.waitForAllResults();
}
```

a)

```
int lower, upper, result;
procedure int checkprime(int a){…}

lower := rangeLowerBound;
upper := rangeUpperBound;

while(low<high){
   result:=checkprime(low);
   if(result#0){<<result;}
   lower++;
}
```

b)

Figure 6.7.: Code example for Workload Partitioning. a) shows the library call in Java. A Tasklet for primes computation is created and performance-aware microtasking is activated. b) shows the respective C-- code that contains the two keywords to support workload partitioning.

6.4.3. Workload Partitioning

The workload partitioning implementation encompasses the four algorithms that were presented in the design chapter: *automatic partitioning*, *performance-aware partitioning* (PAP), *microtasking* (MT), and *performance-aware microtasking* (PAM). For this implementation, several components of the core Tasklet system have been modified.

The Java library is extended with a method to pass information about the data structure of a task to the Tasklet system. The design proposed three different API modes: *range*, *set*, and *runs*. In the prototype, the *range* and *runs* modes are implemented. The *range* partitioning library method has five parameters: *start*, *end*, *minSplit*, *envHet*, as well as *taskIrreg*:

```
TaskletBundle.setPartitioning(int start, int end, int minSplit,
                boolean envHet, boolean taskIrreg);
```

Analogous to this integer version, a version for float values is part of the prototype. Figure 6.7 shows an example code of the workload partitioning. The *start* and *end* parameter describe the range of the data a task is working with. The *minSplit* parameter defines the smallest possible granularity a task can be split into in terms of its data. In the example in Figure 6.7, this parameter is equal to one. The fourth parameter determines if the execution should take heterogeneous environments into account. In case this parameter is set to *true*, the PAP is enabled. If the last parameter is enabled, the approach copes with irregular task structures and activates MT. Once both parameters are set to true, the PAM is enabled and heterogeneous environments and irregular task structures are handled. More exhaustive code examples can be found in Appendix C.

As seen in Figure 6.7 b), the Tasklet language has two additional keywords to realize workload partitioning: *rangeLowerBound* and *rangeUpperBound*. Transparent for the developer, multiple Tasklets are initialized when the partitioning is used. Even with *envHet* and *taskIrreg* set to false, the automatic partitioning mechanism splits up the workload in several equally-sized Tasklets and randomly assigns them to providers in the system. Each of these Tasklets computes a different part of the overall task and, therefore, has to be parameterized individually. The C-- code must be written such that this parameterization can be done during runtime. As a solution, the two keywords work as placeholders, which are initialized with the respective values later on. In the Tasklet factory, the byte code is compiled with these placeholders for a late binding between Tasklets and data. Further, the Tasklet closure is assembled. This process is done only once and the Tasklet is forwarded to the local orchestration. The orchestration then requests resources from the broker. Depending on the setting that the developer made via the library method call, the orchestration decides on a partitioning strategy. If PAP and MT are deactived, the orchestration considers the number of resource providers that were assigned by the broker for the automatic partitioning. The Tasklet is duplicated respectively and for each duplicate, the partitioning parameters are set individually such that each Tasklet computes another part of the task. In case of automatic partitioning, all parts have the same size.

If MT is enabled exclusively, each Tasklet is further subdivided into so called microtasks considering the *minSplit* parameter as the smallest possible granularity. The number of microtasks is not necessarily as small as possible, but is determined by the current context. In the current prototype, the number of microtasks can be determined by the developer manually. Before the orchestration forwards the Tasklets, it shuffles the microtasks among all Tasklet based on a deterministic function that is invertible. By doing so, the number of Tasklets does not change. If PAP is activated exclusively, the aggregated performance index P_{agg} of the assigned providers is computed. Based on that and the performance index p_i

of each provider, the individual share r_i of the task is calculated, as shown in Equation 6.1. The orchestration employs this information and the data structure information given by the developer to assign each provider a respective share.

$$P_{agg} = \sum_{i=1}^{n} p_i \qquad r_i = \frac{p_i}{P_{agg}} \tag{6.1}$$

When MT and PAP are activated, the hybrid PAM algorithm is applied. Therefore, not the parameter range is used to tailor the computational shares for each provider, but the number of microtasks that each provider has to compute. Based on their individual performance p_i, the aggregated performance P_{agg}, and the total number of microtasks, each provider gets a certain amount of microtasks assigned. As an example, if provider A is twice as fast as provider B, A computes twice as many microtasks as B.

Next, the orchestration forwards the Tasklets to the providers accordingly and each TVM inserts the parameters into the byte code. In case MT is enabled, each Tasklet encompasses several non-sequential parts of a task. The TVM runs each microtask individually by applying a reset after each one. The results are concatenated and provided with explicit partitioning data. After execution, the results are sent back to the orchestration of the resource consumer. If MT or PAM is enabled, the results have to be re-ordered, since the MT algorithm shuffles microtasks among the Tasklets. Therefore, after all results were delivered, the inverted deterministic shuffle function is used to bring the results in the correct order.

6.5. Summary

This section presented the implementation of the prototype of the Tasklet system and the edge support layer. The implementation encompasses the entire design from Chapter 5 and consists of the Tasklet system core, the platform support, and the edge-centric components. All parts are integrated and can cooperate within a single distributed computing system to share computational capabilities. In the next chapter, the prototype is evaluated exhaustively based on six experiments and a qualitative analysis.

7. Evaluation

After the design and the implementation chapter, the presented prototype is evaluated comprehensively. The evaluation is divided in seven experiments. The structure of the chapter is as follows: First, an evaluation overview in terms of setup, applications, and structure is given. Then, seven experiments are conducted, the results are presented and discussed.

7.1. Overview

The course of the evaluation started with a first large test run in a university computer lab. Thereto, 78 office computers of two different generations were used and contributed their computational performance to the Tasklet system. An application for image rendering and an artificial intelligence application served as the first resource consumers. Since then, the Tasklet system was steadily extended and further evaluated. The presented evaluation is an excerpt of experiments that support the stated research questions. Evaluation results of the initial tests, the Tasklet gateway, and the benefits of quality of computation are out of this thesis's scope.

Although the Tasklet gateway is a key contribution, the results are narrowed down to a qualitative analysis. The reason behind this decision is that the gathered quantitative evaluation results show the plain benefit of computation offloading, which are shown by the basic Tasklet system functionality already. The focus of the Tasklet gateway is on its architecture that facilitates the integration of thin devices in the first place.

Experiment	Objective	Device Types	Page
0	Requirements Evaluation	–	125
1	Tasklet Baseline	PC	127
2	Mobile Tasklet	Smartphone, PC, Laptop	130
3	GPU-Acceleration	PC, GPU	135
4	Platform Comparison	GPU, PC, Smartphone	139
5	Hybrid Scheduling	PC, Cloud	140
6	Task Migration and Partitioning	PC, Laptop, Smartphone	142

Table 7.1.: Overview of evaluation content.

General Setup

Most of the setup is specific for the individual experiment and, thus, described in the corresponding section. All experiments were conducted in real-world testbeds, consisting of heterogeneous office computers, laptops, smartphones, tablets, GPUs, as well as cloud resources. Each experiment has a particular combination of devices. The benchmarks indicate the single core performance of the device by solving a deterministic and standardized problem. This method is simplified and does not consider turbo clocks and behavior that is caused by active or passive processor cooling systems. All evaluation runs were executed at least 50 times if not stated differently and the average values were used. Several applications were developed for Tasklets, including applications for k-means computation, image rendering and processing, face recognition, and option pricing. These applications cover different domains and belong to the group of applications that benefit from offloading, as categorized in Chapter 5.2. Several applications for Tasklets were presented in [62][1] and [162][2]. For the presented evaluation results, mainly five application were used: *option pricing*, *ray tracing*, rendering of an *MBS*, *prime number finder*, and *gray scale filter*. These applications are all conform to the application model.

Structure

The structure of the experiments is as follows: First, the derived requirements from Chapter 3 are evaluated qualitatively. Second, the Tasklet baseline performance is measured by investigating the parallel execution of Tasklets, the remote placement

[1] [62] is joint work with J. Edinger, M. Breitbach, and C. Becker
[2] [162] is joint work with J. Edinger, M. Breitbach, and C. Becker

REQ	**Tasklets**	**Middle-ware**	**TVM**	**Host Lang.**	**QoC**	**Ad-hoc Hybrid**	**Migr.**	**Workl. Part.**
REQ_{F1}	✓	✓	✓	✓	✓	✓		
REQ_{F2}	✓		✓					
REQ_{F3}							✓	✓
REQ_{F4}		✓				✓	✓	✓
REQ_{F5}	✓	✓	✓	✓	✓	✓	✓	✓
REQ_{F6}		✓						
REQ_{F7}	✓	✓	✓					

Table 7.2.: Summary of the requirement evaluation

of Tasklets, and the scalability of parallel Tasklet executions. Third, the execution of Tasklets on mobile devices is examined as well as the execution behavior. Fourth, the GPU-acceleration of Tasklet is tested and the optimal parallelization granularity is identified. Fifth, the three introduced platforms are compared with each other and possible offloading benefits are demonstrated. Sixth, the hybrid scheduling approach is examined and a threshold for a decision between edge and cloud is defined for the used application. Seventh, the shortcomings of edge environments as resource providers are measured. It is shown that task migration and workload partitioning can counteract these effect, making edge devices applicable as elastic resource providers. The content of the evaluation is shown in Table 7.1.

7.2. Experiment 0: Requirement Evaluation

The functional requirements REQ_{F1}-REQ_{F8} from Chapter 3 are qualitatively evaluated in this experiment and summarized in Table 7.2. The first Requirement *Computation Placement* (REQ_{F1}) is tackled by the Tasklet core system presented in Chapter 5.3. It consists of a middleware that provides the construction, distribution, and execution of Tasklets. The middleware includes an integration mechanisms for application via the host language concept and has a well-defined API. For the application programmer, the entire process is transparent. Further, the tailoring of computation is supported by the QoC concept as well as the hybrid scheduling on cloud and edge resources. Based on that, Tasklet execution can be tailored to the application requirements, such as responsiveness or performance.

The requirement *Lightweight Computation Abstraction* (REQ_{F2}) is tackled by the concept of closed units of computation, the so called Tasklets, in combination with their runtime environment – the TVM. Tasklets enclose all elements that are necessary for their remote execution, namely, logic, parameters, QoC goals, and data. They are executed on a TVM, which abstracts plain computation power of otherwise heterogeneous devices, including PCs, smartphones, tablets, and GPUs.

The *Edge Support* (REQ_{F3}) requirement is supported by the task migration and workload partitioning mechanism. They tackle the characteristics of edge devices, namely, fluctuation, errors, and performance heterogeneity. Further, the edge typically runs application that have irregular task structures, which is covered by microtasking as well.

The fourth requirement *Edge Resource Elasticity* (REQ_{F4}) is addressed by a combination of different artifacts. The ad-hoc scheduling realizes the low latency scheduling of edge devices within proximity, which lays the foundation for elasticity. The characteristics of edge devices naturally counteract the elasticity. Therefore, the two approaches task migration from Chapter 5.4.4 and workload partitioning from Chapter 5.4.5 increase the overall resource efficiency and performance. Based on that, the resource elasticity in edge environments is established.

The fourth requirement *Overcoming Edge Heterogeneity* (REQ_{F5}) has different dimensions. The platform heterogeneity is tackled by the lightweight computation abstraction of Tasklets, the different TVM runtime environments, and the platform specific architectures, such as the gateway or the GTVM. The distribution layer of the orchestration addresses the accessibility heterogeneity, including network connection, fluctuation, and connection errors. The host language concept overcomes the programming language heterogeneity. Application heterogeneity is covered by the QoC concept and the hybrid scheduling. Both take the requirements as well as the characteristics of application into account and execute tasks correspondingly. The irregularity of task structures is also a heterogeneity, which is tackled by the microtasking approach.

Device Type	**CPU**	**Benchmark** (sec)	**Amount**
Amazon EC2 - t2.mirco	1 x 2.9 GHz	3.61	100
Amazon EC2 - m1.xlarge	4 x 2.0 GHz	11.65	10
Amazon EC2 - c4.xlarge	4 x 2.9 GHz	2.89	40
Amazon EC2 - c4.8xlarge	36 x 2.9 GHz	2.87	5
Office Computer (Intel Q6600)	4 x 2.4 GHz	9.26	4
Office Computer (Intel i7-4770)	8 x 3.4 GHz	2.72	1
Intel NUC (Intel i5-4350U)	4 x 2.6 GHz	3.62	1
Nexus 5/7	SD S800/S4	29.9	2
Total			163

Table 7.3.: Overview of the resource provider pool. The resources were geographically distributed across Dublin (Ireland), Frankfurt (Germany), and Mannheim (Germany).

The *Hiding Complexities* (REQ_{F6}) requirement is covered by using a middleware-based approach, since the main task of a middleware is to hide the complexity from developers. However, developers still have a certain control over the execution of Tasklets. The middleware copes with access, location, migration, replication, and failure transparency.

Lastly, the *Unobtrusive* (REQ_{F7}) requirement is tackled by two components: First, the GPU mechanisms that pauses Tasklet executions to let the operating system render the GUI. Second, the mechanisms to drop or pause Tasklets at any point in time, in case local users need resources themselves. With increased fault tolerance, Tasklet drops can be coped without decreasing the application quality, which provides the system with a high degree of flexibility.

7.3. Experiment 1: Tasklet Baseline Performance

In this first experiment, the basic functionality of parallel execution and remote task placement are investigated. First, the setup of the experiment is described and after that the results of the measurements are shown. The following question should be answered: Can Tasklets exploit parallelism and how do they behave when placed on remote providers? This experiment is based on [166][3].

[3][166] is joint work with J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker

Setup

For this part of the evaluation, a pool of heterogeneous Amazon EC2 instances, local office computers, Nexus 7 tablets, Nexus 5 smartphones, and an Intel NUC were used. From this pool, about 163 machines with a total of 518 CPU cores served as resource providers, as shown in Table 7.3. One additional office PC acted as resource consumer and an IBM Blade Center with a Xeon E5345 was used as the resource broker. During the evaluation, more than 2,000,000 Tasklets were executed in 1,400 CPU hours. The application that was used renders an image of a MBS. An MBS is a set of complex numbers that, for a given sequence $z_1 = z_0^2 + c$, does not converge to infinity [27]. The calculation is computationally intensive and thus a candidate to be offloaded. The MBS application is used as a representative for any computationally intensive algorithm that can be split into independent subroutines and has few input and output data. Hence, during the evaluation, the MBS is split up into several Tasklets and executed in parallel. As the measure, the Tasklet bundle turnaround time for the computation of the same image section of a 640×480 pixel MBS is used.

Results

Since modern computing systems have multiple CPUs, splitting up the workload on a local device might result in a faster execution. First, this evaluation determines how suitable the Tasklet middleware is to support parallel computation. The Tasklet system provides a convenient way for the programmer to issue multiple parametrized Tasklets. The MBS image is split up into sets of lines and the time is measured until the last part of the Tasklet bundle arrives at the consumer application. The number of parallel Tasklets depends on the number of physical cores to see how the different devices perform. Figure 7.1 shows the results of the parallel executions. The time decreases with the number of physical cores. Once the parallelization equals the number of cores there is no further improvement. There are two phenomena to be pointed out: First, the improvement of physical cores is greater then for logical cores. Second, for the large EC2 instance, the improvement is minimal between 10 and 16 cores but peaks when the number of cores equals the number of Tasklets instances executed in parallel. In this benchmark, the computation time C still accounts for the majority ($> 95\%$) of the total time T. For further improvements, the Tasklets are placed on remote resource providers.

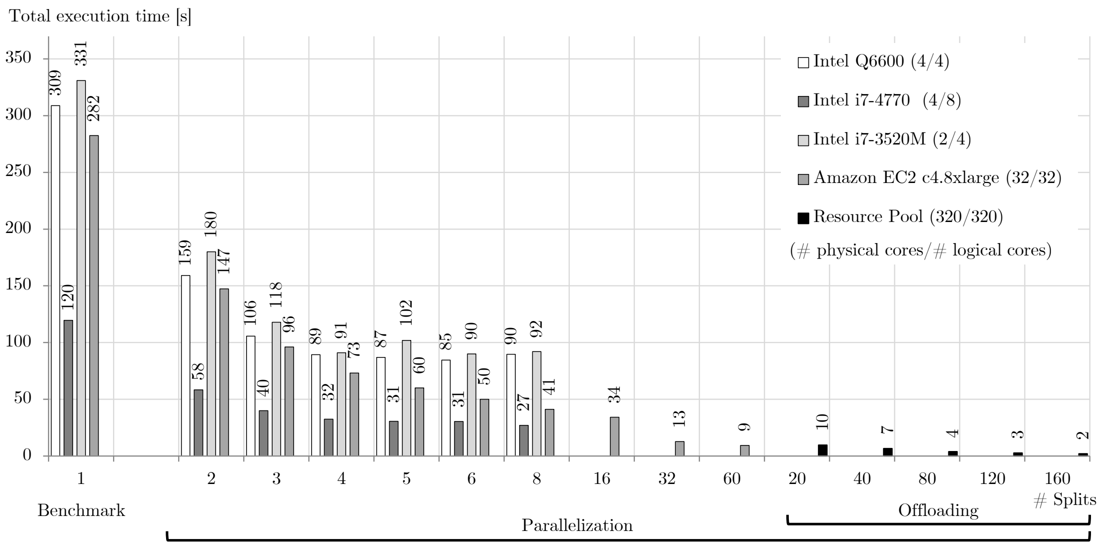


Figure 7.1.: Baseline of the Tasklet system on the example of the computation of a 640×480 pixel MBS. The baseline on the left shows the total execution time of the local computation on four devices (three physical machines and one EC2 virtual machine). The same task is then split up into 2 to 8 chunks for the physical and 2 to 60 chunks for the virtual machine. The results indicate the performance increase for local parallelization. The remote task placement divided the task in up to 160 parts. This further accelerated the execution.

Offloading grants access to a huge pool of resources, however, it comes at the cost of scheduling overhead and network delay. To evaluate how far distribution can improve the applications, the computation of the same MBS image is split up further into even finer granularity between 20 to 160 single Tasklets. For this test, remote resources are used exclusively. Again, each Tasklet has to be compiled, scheduled, executed, and the results have to be sent back to the application. Figure 7.1 shows that remote task placement can significantly benefit the total turnaround time of the MBS application.

Device Name	Operating System	Processor	Frequency (MHz)	Cores (P/L)	Benchmark (Seconds)
HTPC	Windows 10 64Bit	AMD Phenom II X4 965 x86	3400	4/4	6.86
Ultrabook	Windows 7 64Bit	Intel Core i5-3317U x86	1700	2/4	13.58
Office PC	Windows 7 32Bit	Intel Core2Duo E7400 x86	2800	2/2	7.08
Google Nexus 5	Android 5.1.1	Qualcomm Krait 400 ARM	2300	4/4	29.98
Sony Xperia Z1	Android 5.1.1	Qualcomm Krait 400 ARM	2200	4/4	30.06
Samsung Galaxy S3	Android 4.4.2	ARM Cortex-A7	2300	4/4	68.25
Elephone P2000	Android 4.4.2	ARM Cortex-A7	1700	8/8	37.71
Amazon Kindle Fire	FireOS 4.5.4	ARM Cortex-A9	1500	2/2	42.04
Google Nexus One	Android 2.3.1	Qualcomm Scorpion ARM	1000	1/1	69.34

Table 7.4.: Device landscape of the real-world testbed

7.4. Experiment 2: Mobile Tasklet Performance

In this section, the performance of Mobile Tasklets is evaluated. Therefore, different characteristics, such as the performance, network influence, and energy consumption, are examined. This section is based on [163][4] and [165][5].

Setup

For the evaluation, two applications are used. The first application – the *prime number finder* – determines all prime numbers within a given interval. The purpose of this application is to represent all computationally intensive applications that only require the transfer of a small amount of data. The second application is a *gray scale filter* application. This application takes a colored image as input and creates a gray-scale version of it. In contrast to the primes application, this application is data-centric with small computational complexity.

The real world testbed consists of a set of heterogeneous devices that are shown in Table 7.4. The main mobile device for the evaluation is a Google Nexus 5. Throughout the evaluation, the following parameters were changed: the *number of devices* that are involved, the *number of splits* of a Tasklet, and the *network speed* that is assumed for all devices. By changing the number of splits, the overall problem size is constant, but it is shared among multiple Tasklets that are computed in parallel. The problem size of the primes application is static with a range from $90,000$ to $100,000$.

[4][163] is joint work with J. Edinger, T. Borlinghaus, J. M. Paluska, and C. Becker
[5][165] is joint work with J. Edinger, J. Eckrich, M. Breitbach, and C. Becker

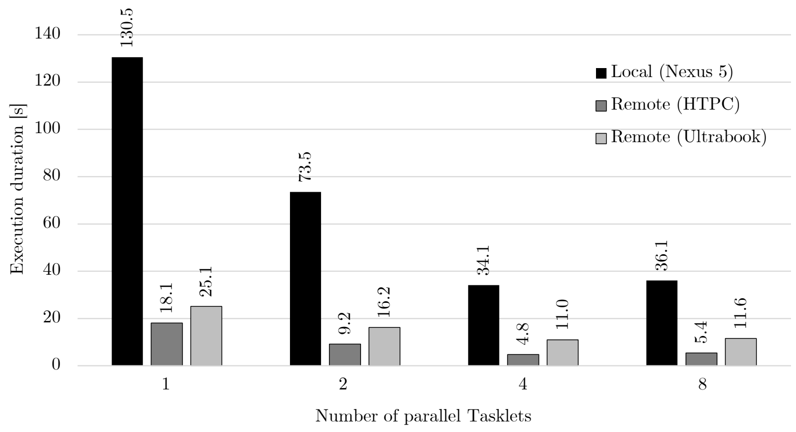


Figure 7.2.: Response times of mobile devices for the primes application. The Tasklet executions on the Nexus 5 are local and on the other two devices are remote, thus the network times are included.

Results

The section answers the following questions systematically: (1) How do mobile devices perform local Tasklet executions? (2) Can mobile devices benefit from remote Tasklet placement? (3) What is the influence of the network connection and device temperature? (4) How does the execution of Tasklet influence the energy consumption of mobile devices?

Performance of Mobile Devices

Figure 7.2 shows the turnaround times for parallel Tasklet executions on the Nexus 5 and on two remote devices – the HTPC and the Ultrabook. For that, the range of Tasklet splits is between 1 and 8. The findings of these experiment show a similar behavior of mobile devices compared with the setup in experiment 1. Splitting up the workload into two Tasklets cuts the turnaround time in half compared to a single Tasklet execution. This holds true for every device. Moreover, splitting up the workload into four Tasklets cuts the turnaround time in again half on the HTPC. The ultrabook, however, is only able to speed up the execution by about 30% when splitting up the workload into four instead of two Tasklets. Increasing of the number of Tasklets further than eight does not influence the execution time on any device. This means that increasing the number of Tasklet splits is only beneficial if a device posses enough physical processor cores.

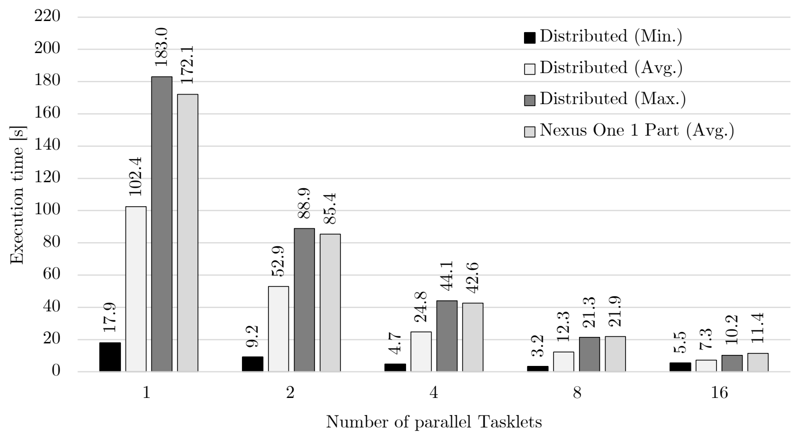


Figure 7.3.: Turnaround times for mobile offloading Tasklets. All devices from Table 7.4 contribute their computational resources. Not only the average is shown, but also the minimum and maximum execution times. In addition, the average execution time of the slowest device, the Nexus One, is depicted.

Mobile Task Placement

To evaluate the offloading feasibility, the complete set of devices from Table 7.4 create a heterogeneous computing environment. Figure 7.3 shows the results. The combination of these devices offers 31 physical cores in total. For this evaluation, the Hyper-Threading Technology has been disabled on the respective processors. The standard deviation of this measurement is much higher than in the previous results, which is referable to the heterogeneity of the computing environment and even-sized splits of Tasklets. Further, the scheduling is random and does not choose devices by execution speed. Hence, in the best case, the scheduler picks the HTPC, which reduces the computation time to one-tenth compared to the slowest device in the environments, which is the Nexus One. This evaluation shows the problems of bottlenecks, meaning, that the computational weakest device determines the total execution duration. Splitting up the task in smaller parts may reduce the effect, but the more Tasklets are in the system, the more likely the slowest device is selected. To cope with highly heterogeneous device landscapes, the automatic task partitioning is evaluated later.

Generation	Technology	Down	Up
2G	GSM GPRS	115 Kbit/s	115 Kbit/s
2G	GSM EDGE	237 Kbit/s	237 Kbit/s
3G	UMTS HSPA	14.4 Mbit/s	5.8 Mbit/s
3G	GSM EDGE-Evo.	1.6 Mbit/s	0.5 Mbit/s
4G	HSPA+	21-672 Mbit/s	5.8-168 Mbit/s
4G	LTE	100-300 Mbit/s	50-75 Mbit/s

Table 7.5.: Network bandwidths

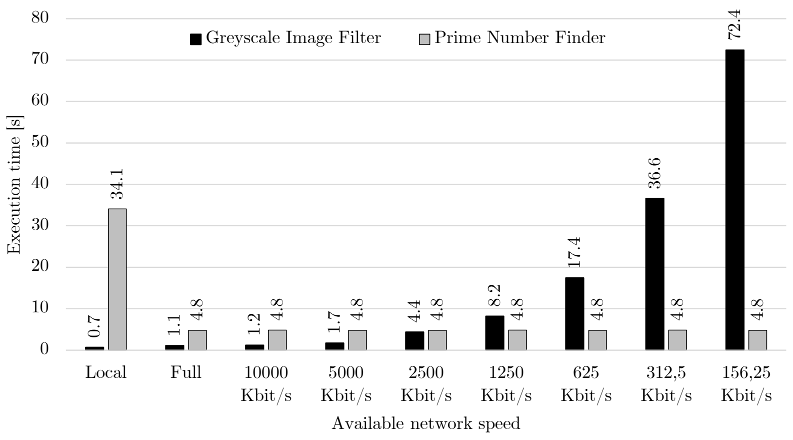


Figure 7.4.: Execution times for the gray-scale filter and the primes application with altering network speeds.

Influence of Network and Temperature

So far, the network connection was not considered in the evaluation. For the primes application, offloading is the best choice in terms of performance. The reasons are that resource providers have a higher performance compared to the local performance of the mobile device and only little data has to be transferred. In contrast, the gray-scale filter represents a task with a larger amount of data and less computation. As a result, such tasks highly depend on the quality of the available network and its bandwidth. Table 7.5 shows today's theoretical available network standards and their maximum bandwidth, which are used for this evaluation.

Figure 7.4 illustrates the performance of both applications in relation to the available bandwidth. For this evaluation, the task is split into four Tasklets. The prime number finder is rather independent from the available bandwidth, since only a little amount of data needs to be transferred. Contrarily, the image greyscale filter highly depends on the available data connection. The two applications are used to show the two extremes in terms of computation- and data-intensity ratio. Other applications are in between these two. As a result, the decision to place a computation remotely always depends on at least these three factors: computational intensity, data intensity, and the effective network speed between a consumer and the provider.

Other context variables may influence this decision as well. The temperature of mobile devices has an influence on their computational capabilities. Most of them are equipped with an ARM processor using a temporal overclocking or downclocking mechanism in case of high temperatures or low battery power. Especially during the relatively short benchmark, the processors activate their overclocking mechanism and work with full performance. During the long runs of evaluation, the temperature of the devices is increased and slows down the Tasklet execution. To reconstruct this effect, Tasklet executions were done on a Moto G4 Plus under different device temperatures. Therefore, the devices was cooled down to 5°C (41°F) and a long running Tasklet over 37.26 seconds was executed. The same Tasklet was executed on a device that was warmed up to 45°C (113°F). This resulted in a runtime of 39.19 seconds. As a result, the temperature differences within that range do not have a major influence on the execution speed, but are noticeable. When approaching the maximal temperatures, the CPU speed is further reduced and the effect is enhanced.

Energy Consumption of Mobile Tasklets

The battery life time is an important context for mobile devices. Thus, the evaluation examines the influence of the Mobile Tasklet system and the local broker on the battery consumption. For this setting, four Moto G4 Plus smartphones formed an ad-hoc network by means of the local broker. Figure 7.5 shows 60 minutes runtime of three different setups. The benchmark line is an idle device with no additional applications running. After that, the energy consumptions of the local broker without the execution of Tasklets is measured. A local broker instance running as client consumes roughly the same energy as a group owner. Hence, both are represented with a single line in the graph. However, when

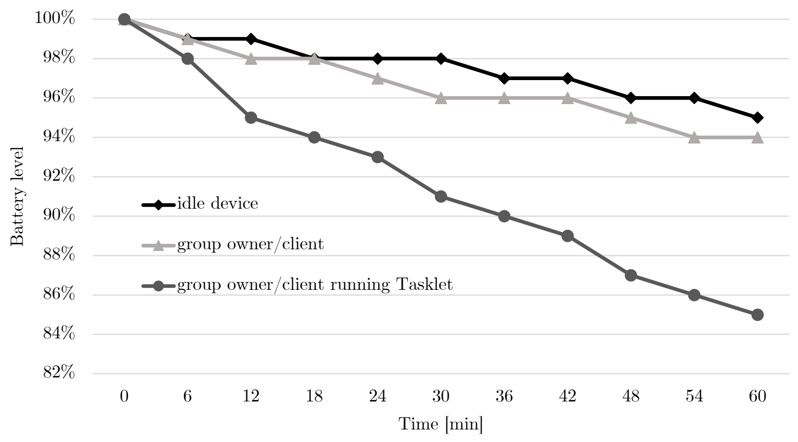


Figure 7.5.: Comparison of the energy consumption. The baseline shows the energy consumption of a Moto G4 Plus without any additional software running for 60 minutes. The other two lines present the mobile energy consumption while running the local broker system and executing Tasklets respectively.

starting the Tasklet system and executing Tasklets, the energy consumption rises substantially. Consequently, the strategy for Tasklet allocation on battery powered devices is crucial and is considered by the utility function from the design chapter.

7.5. Experiment 3: GPU-Acceleration Performance

In this chapter, the performance of GPUs is evaluated including the execution performance, the system usability, and the caching strategies. This part of the evaluation is based on [161][6].

Setup

All tests are executed on a personal computer with a NVIDIA GTX 750 with 2 GB of memory and 512 Cuda processing cores running OpenCL version 1.2. The computer is equipped with an Intel Core i5-2500k with 3.3 GHz, 8 GB of RAM, and Windows 8.1. For this evaluation, the primes application is used with three different problem sizes: (1) small: $0-1,024$, (2) medium: $0-10,240$, and (3) large $0-102,400$. The problem is split into smaller units on the GPU, called Tasklet threads. The size of the Tasklet threads determines the fraction of the problem

[6][161] is joint work with J. Edinger, and C. Becker

that the Tasklet has to compute. For example, a Tasklet thread size of five means that each thread validates five prime numbers. Tasklet threads are grouped into so called work groups, each of which is executed on one streaming multiprocessor. The test investigates various granularities of parallelization, ranging from 1 to 20 items per Tasklet thread and 8 to 512 Tasklet threads per work group. All runs were executed 100 times and the average values were used. This evaluation focuses on the computation times C plus the time for copying the data from the host CPU to the GPU as well as marshaling and unmarshaling operations. The communication times are neglected, as this evaluation focuses on the computational performance.

Results

The following questions are answered successively: (1) How does the degree of parallelism influence the execution performance? (2) Does the work group size affect the performance? (3) How does GPU-accelerated task execution affect the usability of the system? (4) What is the influence of different caching strategies?

Exploiting Parallelism

In the first measurements, the optimum for parallel executions on the GPU is identified. Therefore, two parameters can be adapted: the work group size and the Tasklet thread size. Figure 7.6 a) shows the results for a large problem size and indicates that a fine-grained parallelism improves the execution time. Checking two numbers per Tasklet thread yields the best results and is nearly twice as fast as computing a single item. On the other hand, increasing the Tasklet thread size raises the execution time by at least 13%. Thus, for this example, the trade-off between the parallelization overhead and a coarse granularity is optimal at two items per Tasklet thread. NVIDIA recommends a work group size of 32 items for that particular GPU model. Figure 7.6 a) confirms that recommendation. The optimum for the work group size is between 32 and 128 work group items, with the best result at 32. Reducing the work group size to 16 items impairs the runtime drastically, since the parallel performance of the GPU cannot be utilized, due to idle resources. Also the number of memory accesses increases.

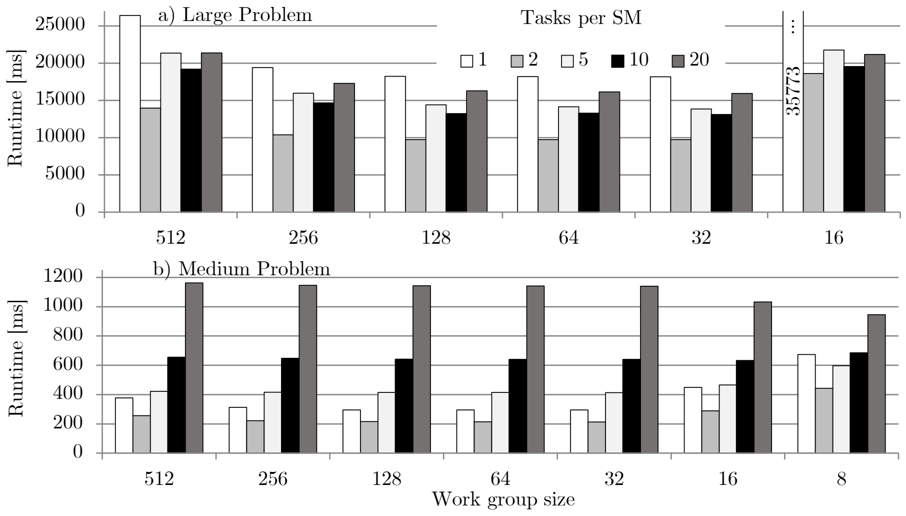


Figure 7.6.: The runtimes of two different problem sizes: a) large and b) medium problem size. For both, different degrees of parallelism were measured by executing different amounts of tasks per steaming multiprocessor. Further, the work group sizes were altered. NVIDIA recommendation for the GTX 750 is a work group size of 32, which is confirmed by the evaluation. Further, a Tasklet thread size of two is optimal.

The results for a medium problem size are similar, as Figure 7.6 b) shows. However, the influence of changing the work group size has a smaller effect. Still, the thread size has a larger influence and the optimum is two items per thread with a work group size of 32. Doubling the thread size increases the runtime by nearly 50%, because the degree of parallelism is decreased. This leads to a situation where some of the 512 GPU cores are idle, while others execute large chunks of the problem. This effect is smaller with the larger problem size, since more work is available. Further, the effect of the irregular task structure of the prime finder increases with the higher range, which leads to uneven execution times. While work group sizes smaller than 32 increase the runtime for fine-grained parallelism, the runtime of coarse-grained settings decreases. This is caused by the utilization of the streaming multiprocessors, since checking 20 numbers per Tasklet thread results in a total of 512 kernels, which is equal to the number of GPU cores. In a large work group setting, like 512, only one streaming multiprocessor is used due to the coarse-grained structure. In contrast, when reducing the work group size to 32 for this problem, all processors are utilized. A work group size of 8 is even more beneficial, since the memory access times are hidden more efficiently.

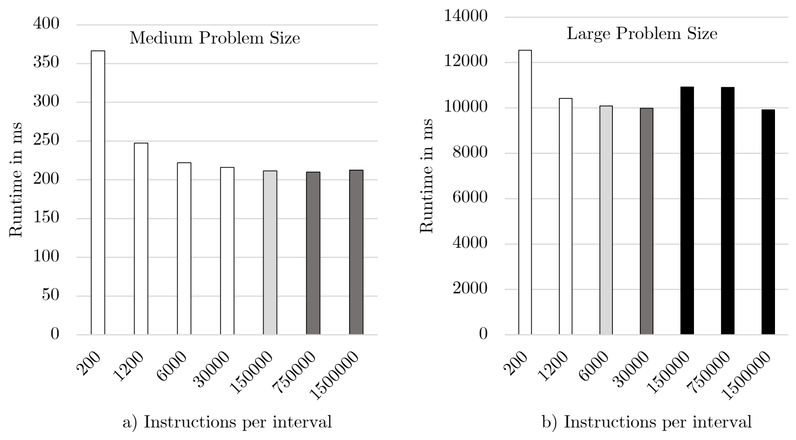
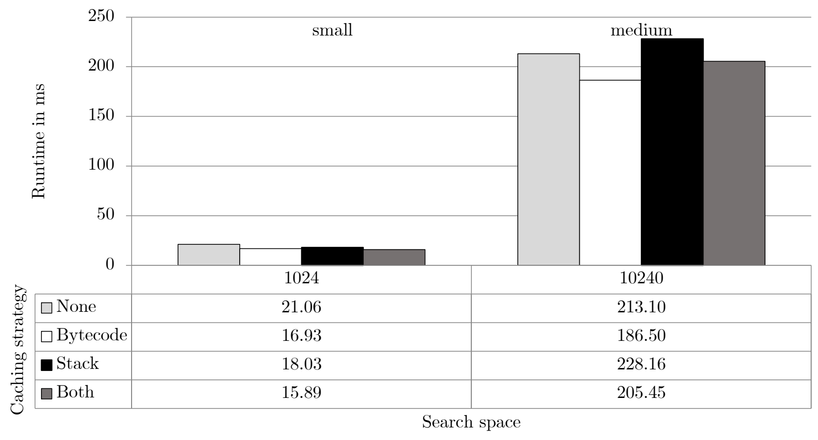


Figure 7.7.: Measurement of GPU user interference. Runtimes for different instruction intervals in between two pauses for a) a medium problem size and b) a large problem size. The colors indicate the subjective perception of the interference for the user. White = no interference, light gray = minor interference, dark gray = major interference, black = system is not usable.

Unobtrusiveness

Figure 7.7 shows the results of the usability measurement. The number of instructions per interval describes how many instructions are executed in between two execution pauses. During these pauses, the operating system renders the GUI. The interval size also influences the Tasklet runtime. The shorter the computation phases between pauses, the longer takes the overall computation. The shade of gray of the bars indicate the subjective perception of the system usability. As stated above, the execution of Tasklets should not compromise the use of the system. For a medium problem size computing 30, 000 instructions is appropriate and has no usability interference. However, the runtime improvement between 6, 000 and 30, 000 is neglectable. For a large problem size, the usability is different, since the overall GPU utilization is much higher. With 6, 000 instructions, the Tasklet execution is already noticeable. After that, it becomes even worse and the system is unusable. However, with the 200 and 1, 200 settings, it is possible to watch a video without any interference. Besides, there are only small runtime improvements after 1, 200 instructions. Under the given circumstances, the setting with 1, 200 is a good option. However, this measurement highly depends on the used hardware and only acts as a rough indication.



Caching strategy	1024	10240
▨ None	21.06	213.10
□ Bytecode	16.93	186.50
■ Stack	18.03	228.16
▨ Both	15.89	205.45

Figure 7.8.: Influence on the runtime of the different caching strategies. *None* implies that no caching is used. *Bytecode* means that only the bytecode of a Tasklet is written into the cache and the *stack* strategy means the same for the stack. With *both*, the stack and the bytecode are cached.

Caching Strategies

Figure 7.8 shows the influence of the different caching strategies for a small and a medium problem size. Caching the bytecode in the local memory benefits the small setting by about 20% and the medium setting by 14%. When caching both, the bytecode and the stack, the execution time can be decreased by 26% for the small problem size. However, it is not beneficial for medium problem sizes to cache the stack in the local memory, due to its small sizes. This effect is even stronger for larger problem sizes. Contrarily, caching the bytecode always leads to faster execution times.

7.6. Experiment 4: Platform Comparison

The final platform measurement compares the execution of three different edge resource types: a Moto G4 Plus smartphone, a PC with a Intel Core i5-2500k CPU, and the NVIDIA GTX 750 GPU. This section is based on [161][7]. Figure 7.9 shows the results and the potential benefit of GPU-Accelerated Tasklet execution in edge environments. Similar to the last experiment, the prime number finder application is used. For a small problem size of 1, 024, the PC shows the best performance,

[7][161] is joint work with J. Edinger, and C. Becker

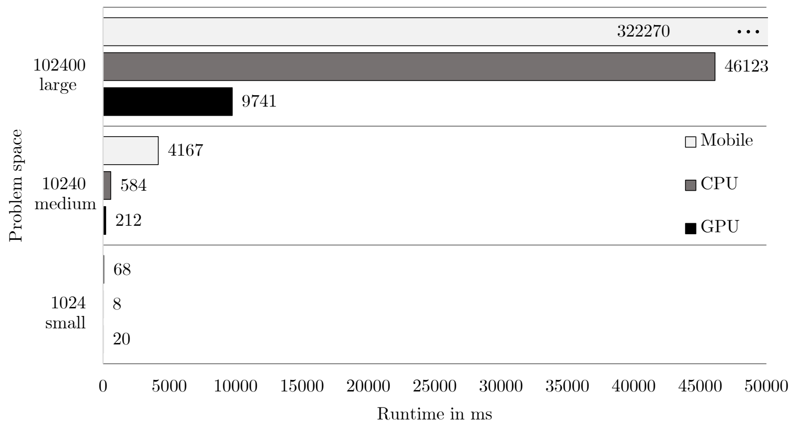


Figure 7.9.: Comparison of execution times on a mobile device, a standard PC, and a GPU for three problem sizes. In the small setting, the GPU-acceleration is not able to compensate the involved overhead. However, for the medium and the large problem, the GPU-acceleration outperforms the other devices by up to 33 times.

since it has a higher CPU clock than the mobile phone. The overhead for the execution on the GPU dominates in this setting. However, the smartphone has the highest execution time, due to the low CPU clock rate. In the medium setting with $10,240$ numbers, the GPU outperforms both by far. The GPU-accelerated execution is approximately 19 times faster compared to the smartphone and nearly three times faster than the CPU. With the large problem size, the GPU can utilize the parallel architecture even more, hence, it is nearly five times faster than the CPU and 33 times faster compared to the smartphone. This evaluation shows the performance heterogeneity in today's computing landscape. Exploiting GPUs in the edge environment can drastically increase the overall performance of a system. Most dedicated GPUs are meant for rendering complex gaming graphics and, thus, are overpowered for standard GUI rendering jobs. In an idle state, GPUs can serve as a powerful edge resource.

7.7. Experiment 5: Hybrid Scheduling Performance

This experiment is based on [165][8]. At this point, cloud resources are integrated into the real-world edge computing testbed. Therefore, three Amazon EC2 t2.micro instances are launched in different data centers as resource providers and remote

[8][165] is joint work with J. Edinger, J. Eckrich, M. Breitbach, and C. Becker

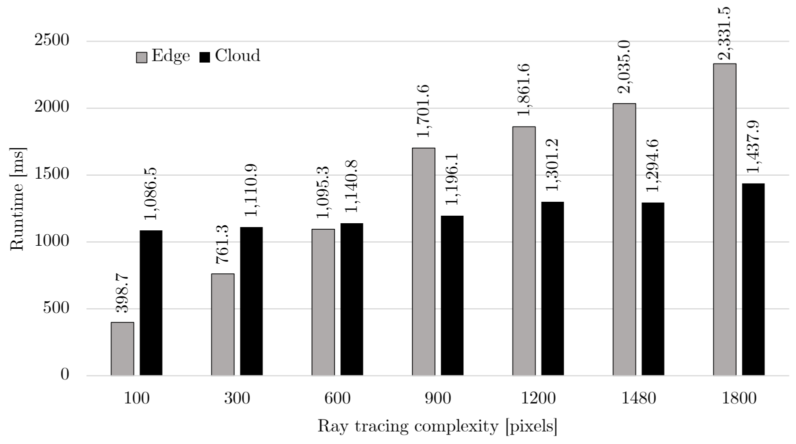


Figure 7.10.: Hybrid scheduling: comparison of edge and cloud computing. In the beginning, the edge outperforms the cloud resources because of the low latency. For 600 pixels, both resources perform similar. After that, the cloud exploits the better performance and compensates the higher communication latencies.

broker. The single core computing performance of the cloud resources is roughly three times higher than performance of the Moto G4 devices. For this evaluation, the ray tracing application is used that generates images with high quality and visual realism. It supports several optical effects, like depth of field, reflection, or chromatic aberration. Therefore, it has a high computational complexity. Offloading benefits ray tracing, since the input parameters are formulas that describe the objects and their features. It is used with different image sizes, ranging from 100 to $1,800$ pixels. Each Tasklet is executed on remote cloud providers as well as on nearby edge devices, using the decentralized scheduling of the local broker which connects the edge devices to an ad-hoc group.

The results are shown in Figure 7.10. Shorter computations (from 100 to 600 pixels) can benefit from the short communication delay of the ad-hoc edge resources. Although the nearby devices are three times slower, the short latency compensates for. For 100 pixels, the edge is 2.73 times faster compared to the cloud. Cloud resources are faster for larger computations and in the 600 pixels setting both perform similar. The gap between edge and cloud resources for a 900 pixel image is already substantial and grows steadily with an increasing number of pixels. Compared to the edge devices, which nearly increase the turnaround times sixfold over the measured range, cloud resource are also more stable. For the largest setting of $1,800$ pixels, the cloud is 1.62 times faster than the edge resources. Both approaches have their merits and the evaluation shows that a hybrid solution can benefit short and long Tasklet executions by scheduling on the most suitable resource. Further, the problem size can be tailored to the environment to optimize the trade-off between low latency and high performance. By increasing the degree of parallelization, the responsiveness of medium or long running Tasklets can also be increased. Hence, in case of a sufficient amount of nearby resources, responsiveness can be maintained even for large Tasklets by splitting them in smaller subtasks and allocating them in the nearby edge.

To further increase the capabilities of edge resources, workload partitioning and task migration are evaluated next.

7.8. Experiment 6: Fault Tolerance and Bottleneck Avoidance

This chapter evaluates the proposed fault tolerance and bottleneck avoidance mechanisms – task migration and workload partitioning – within a real-world testbed. This experiment is based on [164][9]. After introducing the experimental setup, the applications, and the baselines, this section systematically answers the following questions: (1) How do characteristics of edge environments influence execution latencies in distributed computing systems? (2) Can *task migration* handle explicit and implicit system leaves? (3) What is the overhead of *task migration*? (4) Can *performance-aware partitioning* improve the execution latencies in heterogeneous environments? (5) How does *microtasking* influence the task responsiveness? (6) Can a combination of both mechanisms handle highly heterogeneous environments and irregular task structures? (7) What is the overhead of handling heterogeneity and irregularity? (8) How do the mechanisms perform in a medium and highly utilized environment?

Setup

This evaluation uses a real-world testbed with 21 physical devices. As shown in Table 7.6, the testbed consists of smartphones, laptops, and PCs, which corresponds to an average nearby device environment in an office. The last two

[9][164] is joint work with J. Edinger, M. Breitbach, and C. Becker

Device Name	**Operating System**	**Processor**	**Frequency** (MHz)	**Cores** (P/L)	**Benchmark** (Seconds)
2 x PC Fast	Windows 10 64Bit	Intel Core i7-8700k	3700	6/12	2.39
2 x PC Medium	Windows 7 64Bit	Intel Core i5-3500	3300	4/4	5.61
2 x PC Slow	Windows 7	Intel Quad Q6600	2400	4/4	9.26
3 x Dell Laptop	Windows 7	Intel Dual-Core P9400	2400	2/2	10.96
2 x Dell Laptop	Windows 7	Intel Core i5-2520M	2400	2/4	6.46
8 x Motorola G4 Plus	Android 7.0	Snapdragon 617	1500	8/8	22.34
Lenovo T430	Windows 10 64Bit	Intel Core i5-3320M	2600	2/4	(8.58)
Lenovo T410s	Windows 7 32Bit	Intel Core2Duo M520	2400	2/4	(18.10)

Table 7.6.: Device landscape of the heterogeneous real-world testbed. P and L stands for physical and logical processing cores respectively.

Abbreviation	**Description**
BL_{OP}	Baseline Option Pricing
BL_{RT}	Baseline Ray Tracing
HOM_S	Homogeneous Environment Slow
HOM_F	Homogeneous Environment Fast
HET	Heterogeneous Environment; 0% Error
PAP_{OP}	Performance-aware Partitioning; Option Pricing
PAP_{RT}	Performance-aware Partitioning; Ray Tracing
MT	Microtasking; Ray Tracing
PAM_{OP}	Performance-aware Microtasking; Option Pricing
PAM_{RT}	Performance-aware Microtasking; Ray Tracing
BL_{0-30}	Baseline; 0-30% Error; Option Pricing
RE_{0-30}	Reactive Migration; 0-30% Error; Option Pricing
PRO_{0-30}	Proactive Migration; 0-30% Error; Option Pricing
$REPRO_{0-30}$	Reactive and Proactive Migration; 0-30% Error; Option Pricing

Table 7.7.: Evaluation key

devices are used as the resource consumer and the resource broker respectively. All other devices contribute their computational resources as providers. Further, the benchmark indicates the heterogeneous characteristics of the environments. The average benchmark is 13.62 seconds with a standard deviation of 7.76 seconds. Both edge optimization approaches – migration and partitioning – are evaluated within this environment. For the migration part, two different artificial system error rates are induced. In the first setting, 30% of the Tasklets and in the second setting 10% of the Tasklets are dropped. The relation between implicit and explicit Tasklet drops is even. When a Tasklet is dropped, the drop happens after 40%-60% of its overall computation time. In Table 7.7, all abbreviations for the run settings are summarized. To enable comparability, all runs with connection errors are excluded from the results. The overhead measurements are conducted with Wireshark.

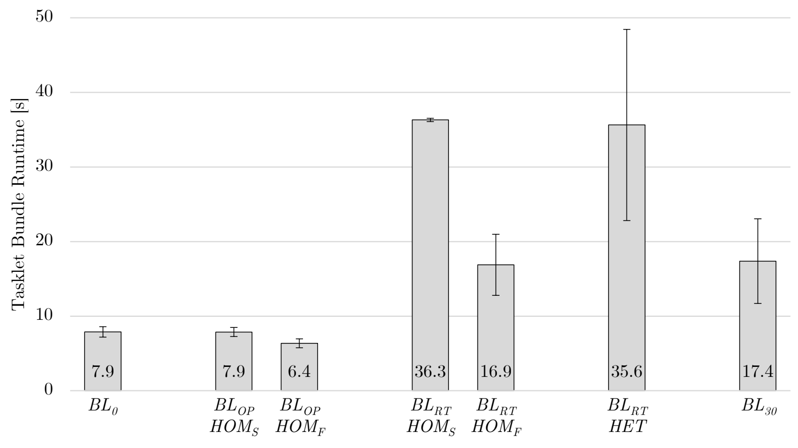


Figure 7.11.: Baseline evaluation BL_0 is done in a homogeneous error free environment that executes homogeneous tasks. Next, slow and fast homogeneous environments are displayed ($BL_{OP}HOM_S/HOM_F$). The following measurements are for heterogeneous tasks ($BL_{RT}HOM_S/HOM_F$). Run $BL_{RT}HET$ shows the full impact of heterogeneity by using the ray tracer application in a heterogeneous environment. The last result BL_{30} introduces errors to the homogeneous system.

For this evaluation, two different applications are used: *option pricing* and *ray tracing*. The first is an *option pricing* algorithm that approximates the Black-Scholes model. This financial algorithm determines a value of a European buy or sell option. The option values is approximated by means of a Monte Carlo-based method, which uses simulations on application basis. This method gains accuracy the more simulation steps are executed. The algorithm is easy to parallelize based on the number of simulations. Thus, the *runs* partitioning setting of the API is used to define the number of parallel simulations per Tasklet. Every run of option pricing is executed with $90,000$ simulations. The second application is the ray tracing algorithm. In this evaluation, the result is a $1,080 \times 720$ image for each computation. The computation is partitioned by means of the image height and use of the *range* partitioning setting. Both applications are split in 12 Tasklets in each setup evaluation. Thus, in case of an even distribution, each Tasklet executes $64,800$ pixels or $7,500$ Monte Carlo simulations for the ray tracing and the option pricing application respectively. These 12 Tasklets are combined in a Tasklet bundle, since they are all required to assemble the application's result.

Results

For the baseline measurements, an error free environment with all devices from Table 7.4 is used. Regarding fault tolerance improvement, the baseline BL_0 in Figure 7.11 marks the lower bound. The influences of errors is indicated by measurement BL_{30}, where the high error rate is applied and the basic reliability mechanism based on heartbeats is used. This mechanism makes use of heartbeats and timeouts to enable Tasklet restarts.

The first error rate setting (BL_{10}) increases the average Tasklet execution time by approximately 18% and the high setting (BL_{30}) by 44% (both not shown in figure). The effect is even stronger when observing the completion time of a Tasklet bundle, which represents one application execution. There, the times increase to about 55% and 120% (BL_{30} in Figure 7.11) respectively, which is accountable to the bottleneck effect that intensifies based on a higher error rate.

The baseline measurement also analyzes the influence of device heterogeneity. Three different settings for this measurement are used: (1) a slow homogeneous HOM_S, (2) a fast homogeneous HOM_F, and (3) a heterogeneous environment HET. The first setting only consists of the slow smartphones. The second setting consists of all devices that have a benchmark faster than 10 seconds and the last setting consists of all devices. The second setting, however, is not entirely homogeneous, but has a certain variance, which can be seen in the deviation of Figure 7.11 HOM_F. Figure 7.11 shows the baseline measures for all three environmental settings executed with the option pricing application, which is homogeneous. For the migration evaluation solely option pricing is used, to exclude task irregularity from the results. Next, the edge optimization mechanisms are applied.

Task Migration

As the baselines indicate, errors have a major influence on execution latencies. Based on the performance measure from the design chapter, the impact of errors can be reduced by different means. Especially in edge computing, devices are unstable and fluctuation of devices is very likely. With the Tasklet migration

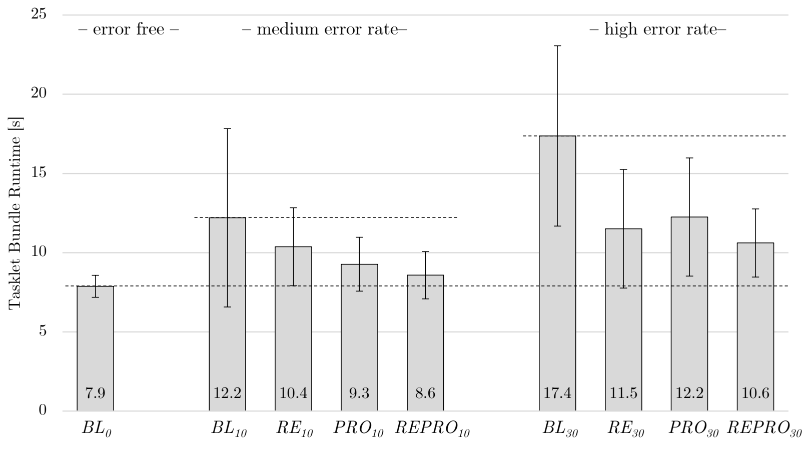


Figure 7.12.: Evaluation result for all Tasklet migration mechanisms in medium and highly erroneous environments compared to an error free baseline (BL_0). In both error settings, reactive (RE), proactive (PRO), as well as hybrid migration ($REPRO$) are applied and the results are compared to the baseline BL respectively.

mechanisms, the loss of computation progress is reduced, as shown in Figure 7.12. Hence, $\sum_{i=0}^{n}(C'_i) + C''$ is reduced to a minimum, which, in an optimal case, is equal to the effective computation time $C_{effective}$.

In a first step, the reactive migration mechanism with a medium error rate (RE_{10}) is applied, which reduces the average execution time by 42% compared to the baselines (BL_0 and BL_{10}), as shown in Figure 7.12. The proactive migration (PRO_{10}) has an even stronger effect and improves the execution time by 67%. This is based on the fact that 50% of the leaves are implicit. For implicit leaves, the reactive approach has no benefit at all, whereas the proactive migration backups and reuses the progress since the last snapshot. Further, the standard deviation of proactive migration is nearly half compared to reactive migration, since outliers are more likely without proactive migration. The combination of both mechanisms ($REPRO_{10}$) outperforms the baseline on average by 84% and the standard deviation is again reduced. Compared to an error free environment (BL_0), the execution times are increased by 8%.

The right part of Figure 7.12 shows the high error rate setting. As expected, the execution times without migration (BL_{30}) are strongly influenced by the error rate and the standard deviation is increased. However, especially for proactive migration (PRO_{30}), the measurements only differ slightly compared to the medium error setting (PRO_{10}). The proactive migration (PRO_{30}) achieves an improvement

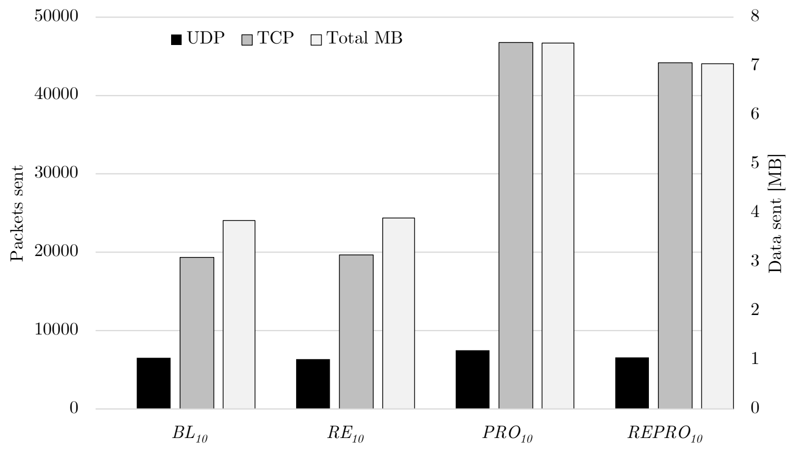


Figure 7.13.: Overhead measurement of Tasklet migration. The baseline BL_{10} without any migration mechanism in the medium erroneous environment is compared to all three migration techniques.

of around 55%. The reactive migration (RE_{30}) improves the reliability baseline (BL_{30}) by 62%. The combination of both migration mechanisms ($REPRO_{30}$) reduces the application runtime by 72%.

Lastly, the overhead of the two migration mechanisms is evaluated. The reactive migration operates under the assumption that system leaves are done gracefully. Thus, the overhead (RE_{10}) compared to the baseline (BL_{10}) is nearly zero, as shown in Figure 7.13. However, proactive migration is also able to cover implicit system leaves. This improves the fault tolerance of the system, but comes at the cost of extra interval snapshot messages (see Figure 7.13, PRO_{10} and $REPRO_{10}$). As a result, the TCP traffic is increased by factor 2.3.

Workload Partitioning

The baseline measurements show that heterogeneity in the edge and task structure irregularity have a major influence on the turnaround times. The completion time of a Tasklet bundle is equal to the completion time of the slowest Tasklet, as formalized in Equation 5.4 in the design chapter on page 73. Especially in edge environments, devices can have enormous performance gaps. Figure 7.14 shows the influence of heterogeneous environments on the overall execution

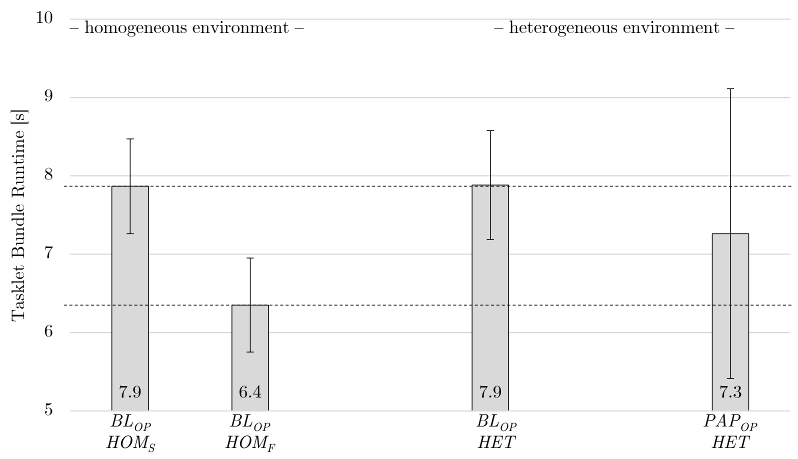


Figure 7.14.: Evaluation results for performance-aware partitioning. The baseline of slow and fast homogeneous ($BL_{OP}HOM_S/HOM_F$) and heterogeneous $BL_{OP}HET$ environments are compared to the optimization of performance-aware partitioning $PAP_{OP}HET$.

time. $PAP_{OP}HET$ displays the benefit of performance-aware partitioning. It achieves an acceleration of about 40% in an heterogeneous environment on average compared to the baselines.

Heterogeneity is not only present in the device landscape, but also in the tasks that are executed. Therefore, the evaluation investigates the behavior of tasks with an irregular structure in homogeneous environments. The results are shown in Figure 7.15 for both, fast and slow homogeneous environments. When applying the *microtasking* approach in both setups, the execution times are improved by 35% for slow ($MT_{RT}HOM_S$) and 21.4% for fast ($MT_{RT}HOM_F$) environments. The variation of these results relates to the fact that the fast environment is not completely homogeneous compared to the slow environment, which consists of exactly the same devices. This can be observed through the deviation indicators of the respective bar diagrams. In settings where environmental heterogeneity is entirely excluded, the full performance of the microtasking approach emerges.

So far, the focus of this experiment is on one kind of heterogeneity at the time. In the next step, environment heterogeneity and task irregularity are combined and the optimization strategies are applied subsequently. The results of that measurement are shown in Figure 7.16. Performance-aware partitioning (PAP_{RT}) achieves an improvement of 31% over the baseline (BL_{RT}), however, only the device heterogeneity is handled by this approach. Next, microtasking (MT_{RT})

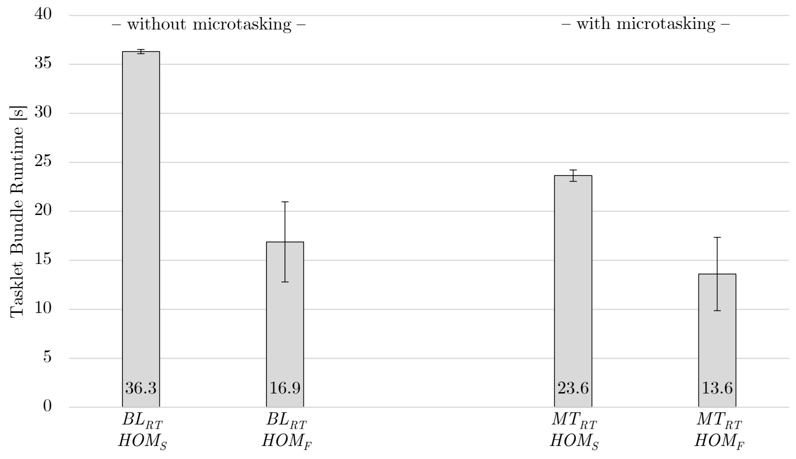


Figure 7.15.: Evaluation results for the microtasking approach. This measurement attempts to exclude the device heterogeneity. However, only the slow setting HOM_S is utterly homogeneous with only Motorola G4 Plus smartphones. The HOM_F setting includes all devices with a benchmark below 10 seconds. The ray tracing application is used, which is highly heterogeneous. The microtasking approach MT_{RT} is compared to the baselines BL_{RT} for both environment settings.

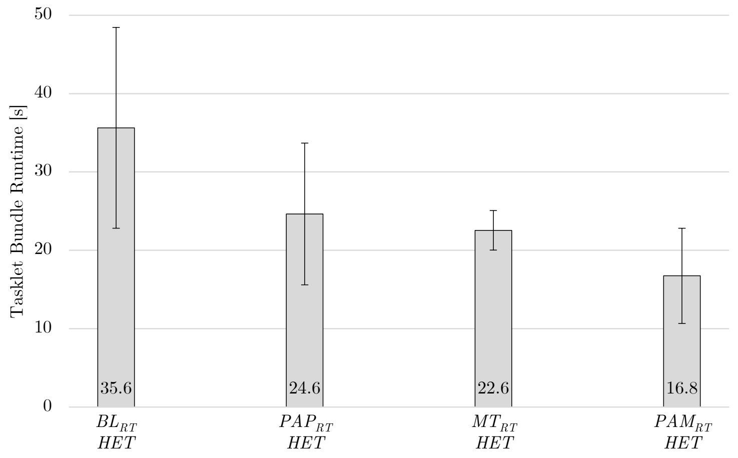


Figure 7.16.: Results for performance-aware microtasking. The heterogeneous ray tracing application is used and all measurements are executed in the heterogeneous computing environment. Starting from the baseline BL_{RT}, the evaluation successively activates performance-aware partitioning PAP_{RT}, microtasking MT_{RT} and finally both, the performance-aware microtasking PAM_{RT}.

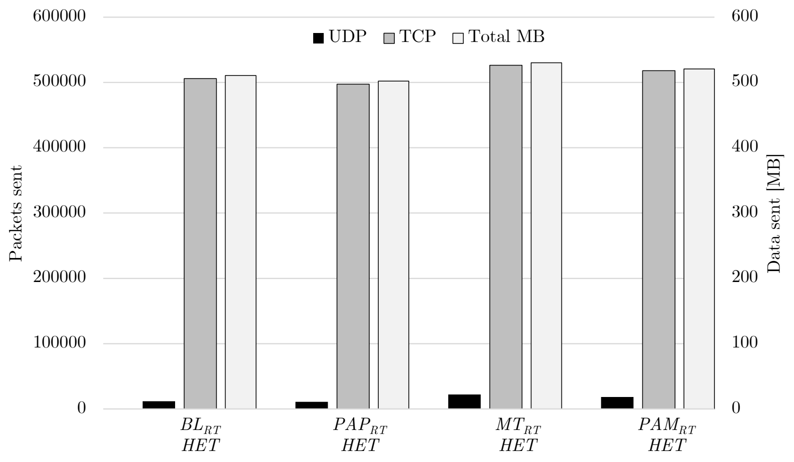


Figure 7.17.: Overhead measurements for the partitioning mechanisms. Based on message and data traffic, the performance-aware partitioning, microtasking, and performance-aware microtasking are compared to the baseline.

is applied under the same circumstances, which outperforms the baseline by 37%. Based on these results it can be concluded, that the task irregularity has a stronger influence in this setting compared to the performance heterogeneity of the devices. Using another application, or another computing environment may change this fact. Finally, both optimizations are combined to the performance-aware microtasking (PAM_{RT}). As a result, it achieves an improvement of the Tasklet bundle completion time of 53%. This outperforms all other measurements substantially. The standard deviation, however, is higher in approaches that apply performance-aware mechanisms.

Next, the number of resource consumers in the system is scaled up and multiple applications issue Tasklet executions concurrently. This measurement shows how the optimization performs in a medium and highly utilized environment. Therefore, the combination of the irregular application structure, the heterogeneous devices, and the performance-aware microtasking ($PAM_{RT}HET$) approach is used. First, two applications ran simultaneously, meaning that the resource utilization of the real-world testbed is approximately at 40%. In that test, no performance decrease was noticeable nor measurable compared to the setting with a single application issuing Tasklets. Second, four applications ran simultaneously, which led to a resource utilization of about 80%. This additional overhead extends the average execution time of the applications by approximately 10%.

Figure 7.17 shows the overhead of all three optimization mechanisms compared to the baseline. Message and traffic overhead is shown. In the evaluation, the overhead of performance-aware partitioning, microtasking, and performance-aware microtasking is neglectable. However, both, option pricing and ray tracing, have only a small amount of input data. One drawback of partitioning is that late binding requires all input data for a Tasklet. Therefore, each Tasklet has to be transferred to the resource provider with the entire input data.

7.9. Summary

This chapter presented the evaluation of the prototype that encompassed seven experiments. First, the qualitative analysis showed that all derived requirements are fulfilled by the approach. In a baseline evaluation, the parallel execution performance of different devices was compared and the offloading capabilities of the Tasklet system illustrated. After that, the characteristics of the Mobile Tasklet approach and the parallel execution of Tasklets on GPUs were examined. The evaluation compared the computational performance of mobile devices, CPUs, and GPUs as well as the benefit of a hybrid scheduling approach. Finally, the evaluation of Tasklet migration and workload partitioning showed that the drawbacks of edge devices can be compensated. The next chapter concludes the thesis and presents directions for future work.

8. Conclusion and Outlook

With trends such as machine learning, virtual reality, and IoT, the computational demand of today's applications grows steadily. In the past decade, these technologies were more and more applied on user-controlled devices. These devices are limited in terms of computing power, which led to the offloading paradigm. Offloading executes computationally intensive tasks on more powerful resources instead of locally. While most of the existing approaches from literature exclusively offload computation to the cloud or other mobile devices, the proposed system exploits user-controlled edge devices of all kinds as resource providers.

Therefore, this thesis presented the Tasklet system and the edge support layer. The resulting system gathers idle computation capabilities of devices at the edge and allows exchange of computationally intensive tasks. The requirements for the design were derived in Chapter 3 and, based on that, a classification for related work was developed in Chapter 4. The literature analysis was conducted and the research gap was identified by means of the classification.

As a solution, Chapter 5 proposed the thesis' design which consists of two major artifacts. The first artifact is the Tasklet system, which laid the foundation for the exchange of computation. Tasklets are fine-grained units of computation that are built as closures, i.e., they include all necessary elements to be executed remotely. The Tasklet system offers an abstraction for computation, a factory approach to assemble Tasklets, an orchestration to allocate them, and a runtime environment for their execution. For the integration of edge devices the edge support layer was proposed, which extends the functionality of the Tasklet system. It integrates several hardware platforms and copes with the challenges of edge devices: fluctuation and heterogeneity. As a solution, the design introduced two Tasklet migration approaches as well as three bottleneck avoidance mechanisms. The migration mechanisms are able to cope with explicit and implicit system leaves at minimal and moderate costs, respectively. The three bottleneck avoidance mechanisms cope with performance heterogeneity of devices as well as with the irregularity of task structures by applying workload partitioning. In combination, the migration and bottleneck avoidance approaches facilitate the elasticity of edge resources. The proposed design was implemented in a full-fledged prototype that was presented in Chapter 6.

Finally, the approach was evaluated comprehensively in Chapter 7, including a qualitative evaluation of the requirements as well as a quantitative evaluation. All functional requirements from Chapter 3 are fulfilled by the presented design and implementation. The non-functional requirements were examined in the quantitative evaluation that was conducted in a real-world testbed with different environment settings. The evaluation compared the execution performances of the different platforms and the hybrid scheduling approach. The integration of GPUs resulted in a substantial performance gain of the edge environment. Further, the evaluation analyzed the benefit of the migration and partitioning approaches that increase the fault tolerance and the bottleneck avoidance. As a result, the performance of the system was improved by 39% and 53%, respectively. Based on these results, the elasticity of edge computing resources can be increased considerably.

The thesis provides the technical foundation to fulfill the vision: 'computation as a common good'. The proposed system offers seamless exchange of computation, a generic interface for applications, and elasticity of resources. Consequently, the next step is to incentivize users to contribute to the system. On the one hand, the participation in well-known systems from the literature, such as BOINC [5] and HTCondor [179], are based on altruism or gamification, which refers to their characteristics: they are single-application system with a specific objective and gather computational power for scientific use cases. On the other hand, in cryptocurrencies users contribute their computational resources for monetary compensation. These two examples show that the type of resource sharing system affects the way users are incentivized. The proposed system lays the ideal foundation for further examination of these incentives, since a variety of applications is covered. A first study is presented in [60][1].

[1] [60] is joint work with J. Edinger, L. M. Edinger-Schons, A. Stelmaszczyk and C. Becker

Outlook

Edge computing is an emerging field of research with many open research challenges. The presented system serves as a foundation for future research:

First, the data management of the Tasklet system offers potential for future research. The current system integrates data into the Tasklet closure. Thus, data is closely coupled to each individual Tasklet. In order to exploit further scheduling options, the data can be separated from the closure and scheduled individually from the Tasklet. By doing so, data can be allocated on resource providers before the actual Tasklet execution takes place. Further, data can be replicated to increase the fault tolerance and enable parallel executions. This strategy improves the execution time as well as the responsiveness. The approach implies several challenges considering the garbage collection, fault tolerance, and data traffic overhead. In [28][2], a data placement system for Tasklets is proposed.

Second, the runtime environment of Tasklets can be improved by optimizing the TVM performance and by integrating the TVM on kernel level. With this approach, a dedicated CPU core is assigned to execute the TVM exclusively. This integration makes the Tasklets independent from the operating system scheduler and facilitates a non-preemptive execution. To issue a Tasklet execution, system calls of the respective operating system can be used.

Third, a future direction for research is the development of a multi-hop scheduling approach for ad-hoc networks. The ad-hoc scheduling currently creates disjunct groups of devices that can share their computational capabilities. In a next step, the scheduling of Tasklets should be possible across the group boundaries to enable a better load balancing. Intermediate nodes in the ad-hoc group facilitate the link to an adjacent group to exchange Tasklets.

Lastly, a large-scale simulation of the approach can identify further directions. So far, the evaluation took place in a real-world testbed that recreates a nearby office scenario as well as a distributed evaluation with cloud resources and roughly 160 resource providers. In the future, a large scale simulation and real world evaluation with more than 1000 providers and consumers as well as multiple resource brokers should be conducted.

[2][28] is joint work with M. Breitbach, J. Edinger, and C. Becker

Bibliography

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, 2018.

[2] S. Abdelwahab, B. Hamdaoui, M. Guizani, and T. Znati. Replisom: Disciplined tiny memory replication for massive iot devices in lte edge cloud. *IEEE Internet of Things Journal*, 3(3):327–338, 2016.

[3] S. Abolfazli, A. Gani, and M. Chen. Hmcc: A hybrid mobile cloud computing framework exploiting heterogeneous resources. In *Proceedings of the 3rd International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2015.

[4] 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.

[5] D. P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings of the Fifth International Workshop on Grid Computing*. IEEE/ACM, 2004.

[6] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[7] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of Job scheduling strategies for parallel processing*. Springer, 2003.

[8] P. Angin, B. Bhargava, and Z. Jin. A self-cloning agents based model for high-performance mobile-cloud computing. In *Proceedings of the 8th International Conference on Cloud Computing*. IEEE, 2015.

[9] 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.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[11] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[12] F. Azzedin and M. Maheswaran. Integrating trust into grid resource management systems. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 2002.

[13] R. D. Ball, V. Bertone, S. Carrazza, C. S. Deans, L. D. Debbio, S. Forte, A. Guffanti, N. P. Hartland, J. I. Latorre, J. Rojo, and M. Ubiali. Parton distributions with lhc data. *Nuclear Physics B*, 867(2):244 – 289, 2013.

[14] M. V. Barbera, S. Kosta, A. Mei, V. C. Perta, and J. Stefa. Mobile offloading in the wild: Findings and lessons learned through a real-life experiment with a new cloud-aware system. In *Proceedings of the Conference on Computer Communications (INFOCOM)*. IEEE, 2014.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Operartion System Review*, 37(5):164–177, 2003.

[16] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *Proceedings of the International Symposium on Parallel Distributed Processing IPDPS*. IEEE, 2009.

[17] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the International Conference on Parallel Processing*, 1995.

[18] F. Berg, F. Dürr, and K. Rothermel. Optimal predictive code offloading. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ICST, 2014.

[19] F. Berman and R. Wolski. The apples project: A status report. In *Proceedings of the 8th NEC Research Symposium*. Citeseer, 1997.

[20] A. Bhattcharya and P. De. Computation offloading from mobile devices: Can edge devices perform better than the cloud? In *Proceedings of the Third International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM, 2016.

[21] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions in Computer Systems*, 2(1):39–59, 1984.

[22] S. Bohez, J. D. Turck, T. Verbelen, P. Simoens, and B. Dhoedt. Mobile, collaborative augmented reality using cloudlets. In *Proceedings of the International Conference on Mobile Wireless Middleware, Operating Systems, and Applications*. Springer, 2013.

[23] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog computing: A platform for internet of things and analytics. In *Proceedings of the Big data and Internet of Things: A roadmap for smart environments*. Springer, 2014.

[24] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. ACM, 2012.

[25] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath. Avatar: Mobile distributed computing in the cloud. In *Proceedings of the International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2015.

[26] S. Bouzefrane, A. F. B. Mostefa, F. Houacine, and H. Cagnon. Engineeringcloudlets authentication in nfc-based mobile computing. In *Proceedings of the International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2014.

[27] B. Branner. The mandelbrot set. In *Proceedings of Symposium in Applied Mathematics*, 1989.

[28] M. Breitbach, D. Schäfer, J. Edinger, and C. Becker. Context-aware data and task placement in edge computing environments. In *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2019.

[29] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[30] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture of a resource management and scheduling system in a global computational grid. *arXiv preprint cs/0009021*, 2000.

[31] R. Buyya, D. Abramson, and J. Giddy. A case for economy grid architecture for service oriented grid computing. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2001.

[32] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1507–1542, 2002.

[33] R. Buyya and S. Vazhkudai. Compute power market: towards a market-oriented grid. In *Proceedings of the International Symposium on Cluster Computing and the Grid*. IEEE/ACM, 2001.

[34] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[35] B. Calder, A. A. Chien, J. Wang, and D. Yang. The entropia virtual machine for desktop grids. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*. ACM/USENIX, 2005.

[36] R. N. Calheiros, C. Vecchiola, D. Karunamoorthy, and R. Buyya. The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds. *Future Generation Computer Systems*, 28(6):861–870, 2012.

[37] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. *Future generation computer systems*, 21(3):417–437, 2005.

[38] S. Chakravorty, C. L. Mendes, and L. V. Kalé. Proactive fault tolerance in large systems. In *Proceedings of the HPCRI Workshop in conjunction with International Conference on High Performance Computing*, 2005.

[39] 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*. Springer, 2006.

[40] A. Chandra and J. B. Weissman. Nebulas: Using distributed voluntary resources to build clouds. In *HotCloud*. USENIX, 2009.

[41] S. J. Chapin, D. Katramatos, J. Karpovich, and A. S. Grimshaw. The legion resource management system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1999.

[42] M. Chiang and T. Zhang. Fog and iot: An overview of research opportunities. *Internet of Things Journal*, 3(6):854–864, 2016.

[43] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.

[44] S. Choi, R. Buyya, H. Kim, E. Byun, M. Baik, J. Gil, and C. Park. A taxonomy of desktop grids and its mapping to state of the art systems. *Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Tech. Rep*, 2008.

[45] S. Choi, H. Kim, E. Byun, M. Baik, S. Kim, C. Park, and C. Hwang. Characterizing and classifying desktop grid. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, 2007.

[46] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the Conference on Computer Systems*. ACM, 2011.

[47] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education, 2005.

[48] 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.

[49] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1998.

[50] S. K. Datta and C. Bonnet. A lightweight framework for efficient m2m device management in onem2m architecture. In *Proceedings of the International Conference on Recent Advances in Internet of Things (RIoT)*. IEEE, 2015.

[51] S. K. Datta, C. Bonnet, and J. Haerri. Fog computing architecture to enable consumer centric internet of things services. In *Proceedings of the International Symposium on Consumer Electronics (ISCE)*. IEEE, 2015.

[52] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *In Proceedings of Operating Systems Design and Implementation (OSDI)*. ACM, 2004.

[53] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[54] S. Delamare, G. Fedak, D. Kondo, and O. Lodygensky. Spequlos: A qos service for bot applications using best effort distributed computing infrastructures. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012.

[55] S. Delamare, G. Fedak, D. Kondo, and O. Lodygensky. Spequlos: A qos service for hybrid and elastic computing infrastructures. *Cluster Computing*, 17(1):79–100, 2014.

[56] A. K. Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[57] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communication and Mobile Computing*, 13(18):1587–1611, 2013.

[58] S. Djilali, T. Herault, O. Lodygensky, T. Morlier, G. Fedak, and F. Cappello. Rpc-v: Toward fault-tolerant rpc for internet connected desktop grids with volatile nodes. In *Proceedings of the Conference on Supercomputing*. ACM/IEEE, 2004.

[59] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, Queen's University Kingston Ontario, 2006.

[60] J. Edinger, L. M. Edinger-Schons, A. Stelmaszczyk, D. Schäfer, and C. Becker. Of money and morals - the contingent effect of monetary incentives in peer-to-peer volunteer computing. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*. ACM, 2019.

[61] J. Edinger, D. Schäfer, and C. Becker. Decentralized scheduling for tasklets. In *Proceedings of the Posters and Demos Session of the International Middleware Conference*. ACM, 2016.

[62] J. Edinger, D. Schäfer, M. Breitbach, and C. Becker. Developing distributed computing applications with tasklets. In *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2017.

[63] J. Edinger, D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker. Fault-avoidance strategies for context-aware schedulers in pervasive computing systems. In *Proceedings of the International Conference on Pervasive Computing and Communication (PerCom)*. IEEE, 2017.

[64] C. Engelmann, H. Ong, and S. L. Scott. Middleware in modern high performance computing system architectures. In *Computational Science – ICCS 2007*. Springer, 2007.

[65] H. Eom, R. Figueiredo, H. Cai, Y. Zhang, and G. Huang. Malmos: Machine learning-based mobile offloading scheduler with online training. In *Proceedings of the International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2015.

[66] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: A generic global computing system. In *Proceedings on the International Symposium on Cluster Computing and the Grid*. IEEE/ACM, 2001.

[67] E. Feller, L. Rilling, and C. Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings. of the International Symposium on Cluster, Cloud and Grid Computing*. IEEE/ACM, 2012.

[68] N. Fernando, S. W. Loke, and W. Rahayu. Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds. *Transactions on Cloud Computing*, pages 1–1, 2018.

[69] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya. Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, 53(3):80–88, 2015.

[70] H. Flores and S. Srirama. Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning. In *Proceedings of the Workshop on Mobile Cloud Computing and Services*. ACM, 2013.

[71] M. J. Flynn. Some computer organizations and their effectiveness. *Transactions on Computers*, 100(9):948–960, 1972.

[72] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[73] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[74] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[75] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *Proceedings of the International Workshop on Quality of Service (IWQOS)*. IEEE, 2000.

[76] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Proceedings of the Grid Computing Environments Workshop*. IEEE, 2008.

[77] 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. *SIGCOMM Computer Communications Review*, 45(5):37–42, 2015.

[78] 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*, 2(1):88–97, 2011.

[79] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In *Proceedings of the International Conference on Supercomputing*. ACM/IEEE, 2005.

[80] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. ACM, 2012.

[81] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. Libwater: Heterogeneous distributed computing made easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13. ACM, 2013.

[82] R. L. Grossman. The case for cloud computing. *IT Professional*, 11(2):23–27, 2009.

[83] S. Guo, H.-z. Huang, Z. Wang, and M. Xie. Grid Service Reliability Modeling and Optimal Task Scheduling Considering Fault Recovery. *Transactions on Reliability*, 60(1):263–274, 2011.

[84] S. Gurun, R. Wolski, C. Krintz, and D. Nurmi. On the efficacy of computation offloading decision-making strategies. *International Journal of High Performance Computing Applications*, 22(4):460–479, 2008.

[85] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the Annual International Conference on Mobile Systems, applications, and services*. ACM, 2014.

[86] K. Habak, M. Ammar, K. A. Harras, and E. Zegura. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *Proceedings of the International Conference on Cloud Computing*. IEEE, 2015.

[87] S. Haider, M. Imran, I. A. Niaz, S. Ullah, and M. A. Ansari. Component based proactive fault tolerant scheduling in computational grid. In *Proceedings of the International Conference on Emerging Technologies*. IEEE, 2007.

[88] 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.

[89] R. Hasan, M. M. Hossain, and R. Khan. Aura: An iot based cloud infrastructure for localized mobile computation outsourcing. In *Proceedings of the International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2015.

[90] M. A. Hassan, K. Bhattarai, Q. Wei, and S. Chen. Pomac: Properly offloading mobile applications to clouds. *Energy (J)*, 25:50, 2014.

[91] M. A. Hassan, M. Xiao, Q. Wei, and S. Chen. Help your mobile applications with fog computing. In *Proceedings of the International Conference on Sensing, Communication, and Networking - Workshops (SECON Workshops)*. IEEE, 2015.

[92] M. Heck, J. Edinger, D. Schaefer, and C. Becker. Iot applications in fog and edge computing: Where are we and where are we going? In *Proceedings of the International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018.

[93] N. R. Herbst, S. Kounev, and R. H. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*. ACM, 2013.

[94] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *International Workshop on Grid Computing*. Springer, 2000.

[95] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile edge computing—a key technology towards 5g. *White Paper*, 11(11):1–16, 2015.

[96] M. T. Huda, H. W. Schmidt, and I. D. Peake. An agent oriented proactive fault-tolerant framework for grid computing. In *Proceedings of the International Conference on e-Science and Grid Computing*. IEEE, 2005.

[97] S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the grid. *Journal of Grid Computing*, 1(3):251–272, 2003.

[98] S. Imai and C. A. Varela. Light-weight adaptive task offloading from smartphones to nearby computational resources. In *Proceedings of the Symposium on Research in Applied Computation*. ACM, 2011.

[99] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the SIGOPS Operating Systems Review*. ACM, 2007.

[100] W. A. Jansen. Cloud hooks: Security and privacy issues in cloud computing. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2011.

[101] B. Javadi, D. Kondo, J.-M. Vincent, and D. P. Anderson. Mining for availability models in large-scale distributed systems : A case study of seti@home. Technical report, Inria Grenoble - Rhône-Alpes, LIG - Laboratoire d'Informatique de Grenoble, 2009.

[102] M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono. On technical security issues in cloud computing. In *Proceedings of the International Conference on Cloud Computing*. IEEE, 2009.

[103] A. Jonathan, M. Ryden, K. Oh, A. Chandra, and J. Weissman. Nebula: Distributed edge cloud for data intensive computing. *Transactions on Parallel and Distributed Systems*, 28(11):3229–3242, 2017.

[104] V. Jungnickel, K. Manolakis, W. Zirwas, B. Panzner, V. Braun, M. Lossow, M. Sternad, R. Apelfrojd, and T. Svensson. The role of small cells, coordinated multipoint, and massive mimo in 5g. *Communications Magazine*, 52(5):44–51, 2014.

[105] T. Justino and R. Buyya. Outsourcing resource-intensive tasks from mobile apps to clouds: Android and aneka integration. In *Proceedings of the International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 2014.

[106] W. Kang and A. Grimshaw. Failure prediction in computational grids. In *Proceedings of the Annual Simulation Symposium*. ACM, 2007.

[107] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. *Cuckoo: A Computation Offloading Framework for Smartphones*, pages 59–79. Springer, 2012.

[108] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the International Conference on Supercomputing*. ACM, 2012.

[109] O. T. T. Kim, N. D. Tri, N. H. Tran, C. S. Hong, et al. A shared parking model in vehicular network using fog and cloud environment. In *Proceedings of the Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2015.

[110] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the International Conference on International Conference on Supercomputing*. ACM, 2013.

[111] T. Kosar and M. Balman. A new paradigm: Data-aware scheduling in grid computing. *Future Generation Computer Systems*, 25(4):406–413, 2009.

[112] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of the INFOCOM*. IEEE, 2012.

[113] S. Kosta, V. C. Perta, J. Stefa, P. Hui, and A. Mei. Clone2clone (c2c): Peer-to-peer networking of smartphones on the cloud. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. ACM, 2013.

[114] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[115] I. Krsul, A. Ganguly, J. Zhang, J. A. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the International Supercomputing Conference*. ACM/IEEE, 2004.

[116] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[117] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[118] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *Proceedings of the International Conference on Autonomic Computing*. ACM, 2009.

[119] Y. W. Kwon and E. Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2012.

[120] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of European Conference on Computer Systems*. ACM, 2009.

[121] M. Lamanna. The lhc computing grid project at cern. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 534(1):1–6, 2004.

[122] X. S. Le, J. C. L. Lann, L. Lagadec, L. Fabresse, N. Bouraqadi, and J. Laval. Cardin: An agile environment for edge computing on reconfigurable sensor networks. In *Proceedings of the International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2016.

[123] 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.

[124] A. Legrand and C. Touati. Non-cooperative scheduling of multiple bag-of-task applications. In *Proceedings of the International Conference on Computer Communications (INFOCOM)*. IEEE, 2007.

[125] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft. Picasso: A lightweight edge computing platform. In *Proceedings of the International Conference on Cloud Networks*. IEEE, 2017.

[126] G. Lewis, S. Echeverría, S. Simanta, B. Bradshaw, and J. Root. Tactical cloudlets: Moving cloud computing to the edge. In *Proceedings of the Military Communications Conference*. IEEE, 2014.

[127] C. Li and L. Li. Utility-based qos optimisation strategy for multi-criteria scheduling on the grid. *Journal of Parallel and Distributed Computing*, 67(2):142–153, 2007.

[128] C. Li, Y. Xue, J. Wang, W. Zhang, and T. Li. Edge-oriented computing paradigms: A survey on architecture design and system management. *Computing Surveys*, 51(2):39:1–39:34, 2018.

[129] Y. Li and Z. Lan. A survey of load balancing in grid computing. In *Proceedings of the Computational and Information Science*. Springer, 2005.

[130] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 2001.

[131] 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*, 23(2):163 – 178, 2007.

[132] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE, 1988.

[133] J. Liu, T. Zhao, S. Zhou, Y. Cheng, and Z. Niu. Concert: a cloud-based architecture for next-generation cellular systems. *Wireless Communications*, 21(6):14–22, 2014.

[134] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, and L. Sun. Fog computing: Focusing on mobile users at the edge. *arXiv preprint arXiv:1502.01815*, 2015.

[135] R. K. Ma and C.-L. Wang. Lightweight application-level task migration for mobile cloud computing. In *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2012.

[136] P. Mach and Z. Becvar. Mobile edge computing: A survey on architecture and computation offloading. *Communication Surveys and Tutorials*, 19(3):1628–1656, 2017.

[137] R. Mahmud, R. Kotagiri, and R. Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[138] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys and Tutorials*, 19(4):2322–2358, 2017.

[139] E. E. Marinelli. Hyrax: cloud computing on mobile devices using mapreduce. Technical report, Carnegie-Mellon University Pittsburgh, School of Computer Science, 2009.

[140] P. Mell, T. Grance, et al. The nist definition of cloud computing. Technical Report 6, National Institute of Standards and Technology, 2011.

[141] A. Mtibaa, A. Fahim, K. A. Harras, and M. H. Ammar. Towards resource sharing in mobile device clouds: Power balancing across mobile devices. *SIGCOMM Computer Communication Review*, 43(4):51–56, 2013.

[142] A. Mtibaa, K. A. Harras, and A. Fahim. Towards computational offloading in mobile device clouds. In *Proceedings of the International Conference on Cloud Computing Technology and Science*. IEEE, 2013.

[143] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-based parallel computing on the internet. In *Proceedings of the European Conference on Parallel Processing*. Springer, 2000.

[144] B. C. Neuman. Readings in distributed computing systems. *Chapter Scale in Distributed Systems, Computer Society*, pages 463–89, 1994.

[145] G. Orsini, D. Bade, and W. Lamersdorf. Computing at the mobile edge: Designing elastic android applications for computation offloading. In *Proceedings of the IFIP Wireless and Mobile Networking Conference (WMNC)*. IEEE, 2015.

[146] G. Orsini, D. Bade, and W. Lamersdorf. Cloudaware: A context-adaptive middleware for mobile edge and cloud computing applications. In *Proceedings of the International Workshops on Foundations and Applications of Self* Systems*. IEEE, 2016.

[147] T. Parr. *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009.

[148] M. Patel, J. Joubert, J. R. Ramos, N. Sprecher, S. Abeta, and A. Neal. Mobile-edge computing. *ETSI White Paper*, 11(1):1–36, 2014.

[149] A. Polze, P. Troger, and F. Salfner. Timely virtual machine migration for pro-active fault tolerance. In *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*. IEEE, 2011.

[150] M. B. Qureshi, M. M. Dehnavi, N. Min-Allah, M. S. Qureshi, H. Hussain, I. Rentifis, N. Tziritas, T. Loukopoulos, S. U. Khan, C.-Z. Xu, and A. Y. Zomaya. Survey on grid resource allocation mechanisms. *Journal of Grid Computing*, 12(2):399–441, 2014.

[151] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proceedings of the Conference on Supercomputing*. ACM/IEEE, 2007.

[152] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Resource failure prediction in fine-grained cycle sharing systems. In *Proceedings of the International Symposium on High Performance Distributed Computing*. IEEE, 2006.

[153] B. P. Rimal, E. Choi, and I. Lumb. A taxonomy and survey of cloud computing systems. In *Proceedings of the International Joint Conference on INC, IMS and IDC*. IEEE, 2009.

[154] R. Roman, J. Lopez, and M. Mambo. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680–698, 2018.

[155] C. Rong, S. T. Nguyen, and M. G. Jaatun. Beyond lightning: A survey on security challenges in cloud computing. *Computers and Electrical Engineering*, 39(1):47 – 54, 2013.

[156] 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*. ACM, 2010.

[157] M. Ryden, K. Oh, A. Chandra, and J. Weissman. Nebula: Distributed edge cloud for data-intensive computing. In *Proceedings of the International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2014.

[158] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing*, 8(4):14–23, 2009.

[159] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani. An open ecosystem for mobile-cloud convergence. *Communications Magazine*, 53(3):63–70, 2015.

[160] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *Pervasive Computing*, 14(2):24–31, 2015.

[161] D. Schäfer, J. Edinger, and C. Becker. Gpu-accelerated task execution in heterogeneous edge environments. In *Proceedings of the International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018.

[162] D. Schäfer, J. Edinger, C. Becker, and M. Breitbach. Writing a distributed computing application in 7 minutes with tasklets. In *Proceedings of the Posters and Demos Session of the International Middleware Conference*. ACM, 2016.

[163] D. Schäfer, J. Edinger, T. Borlinghaus, J. M. Paluska, and C. Becker. Using quality of computation to enhance quality of service in mobile computing systems. In *Proceedings of the International Symposium on Quality of Service*. IEEE, 2017.

[164] D. Schäfer, J. Edinger, M. Breitbach, and C. Becker. Workload partitioning and task migration to reduce response times in heterogeneous computing environments. In *Proceedings of the International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018.

[165] D. Schäfer, J. Edinger, J. Eckrich, M. Breitbach, and C. Becker. Hybrid task scheduling for mobile devices in edge and cloud environments. In *Proceedings of the International Workshop on Smart Edge Computing and Networks in conjunction with the International Conference on Pervasive Computing and Communication (PerCom)*. IEEE, 2018.

[166] D. Schäfer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker. Tasklets: "better than best-effort" computing. In *Proceedings of the International Conference on Computer Communication and Networks (ICCCN)*, 2016.

[167] D. Schäfer, J. Edinger, S. VanSyckel, J. M. Paluska, and C. Becker. Tasklets: Overcoming heterogeneity in distributed computing systems. In *Proceedings of the Workshop on Edge Computing on ICDCS*. IEEE, 2016.

[168] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*. IEEE, 1994.

[169] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura. Serendipity: Enabling remote computing among intermittently connected mobile devices. In *Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing*. ACM, 2012.

[170] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *Internet of Things Journal*, 3(5):637–646, 2016.

[171] J. Shiers. The worldwide lhc computing grid (worldwide lcg). *Computer Physics Communications*, 177(1):219 – 223, 2007.

[172] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. Gridbot: execution of bags of tasks in multiple grids. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009.

[173] J. Sonnek, A. Chandra, and J. Weissman. Adaptive reputation-based scheduling on unreliable distributed infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1551–1564, 2007.

[174] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the International Symposium on High Performance Distributed Computing*. ACM, 2008.

[175] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman. Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Proceedings of the Symposium on Computers and Communications (ISCC)*. IEEE, 2012.

[176] I. Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *Proceedings of the Telecommunication Networks and Applications Conference (ATNAC)*. IEEE, 2014.

[177] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science and engineering*, 12(1-3):66–73, 2010.

[178] H. Takabi, J. B. D. Joshi, and G. Ahn. Security and privacy challenges in cloud computing environments. *Security Privacy*, 8(6):24–31, 2010.

[179] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[180] P. Townend and J. Xu. Fault Tolerance within a Grid Environment. *Engineering and Physical Sciences Research Council (EPSRCA)*, 1:1–4, 2005.

[181] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili. Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges. *Communications Magazine*, 55(4):54–61, 2017.

[182] R. Trivedi, A. Chandra, and J. Weissman. Heterogeneity-aware workload distribution in donation-based grids. *The International Journal of High Performance Computing Applications*, 20(4):455–466, 2006.

[183] G. L. Valentini, W. Lassonde, S. U. Khan, N. Min-Allah, S. A. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej, H. Li, A. Y. Zomaya, C.-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, and P. Bouvry. An overview of energy efficiency techniques in cluster computing systems. *Cluster Computing*, 16(1):3–15, 2013.

[184] M. Van Steen and A. S. Tanenbaum. *Distributed systems: principles and paradigms*. Prentice-Hall, 3 edition, 2017.

[185] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[186] C. Vecchiola, X. Chu, and R. Buyya. Aneka: a software platform for .net-based cloud computing. *High Speed and Large Scale Scientific Computing*, 18:267–295, 2009.

[187] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Storn. Spawn: A distributed computational economy. *Transactions on Software Engineering*, 18(2):103–117, 1992.

[188] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *Proceedings of the Conference on Supercomputing*. ACM/IEEE, 2008.

[189] Q. Wang and K. Wolter. Accelerating task completion in mobile offloading systems through adaptive restart. *Software and Systems Modeling*, 15:1–17, 2016.

[190] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang. A survey on mobile edge networks: Convergence of computing, caching and communications. *Journal on Access*, 5:6757–6779, 2017.

[191] M. Weiser. The computer for the 21 st century. *Scientific American*, 265(3):94–105, 1991.

[192] J. B. Weissman, P. Sundarrajan, A. Gupta, M. Ryden, R. Nair, and A. Chandra. Early experience with the distributed nebula cloud. In *Proceedings of the International Workshop on Data-intensive Distributed Computing*. ACM, 2011.

[193] C. Weng and X. Lu. Heuristic scheduling for bag-of-tasks applications in combination with qos in the computational grid. *Future Generation Computer Systems*, 21(2):271–280, 2005.

[194] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational grid. *The International Journal of High Performance Computing Applications*, 15(3):258–281, 2001.

[195] F. Xhafa and A. Abraham. Computational models and heuristic methods for grid scheduling problems. *Future generation computer systems*, 26(4):608–621, 2010.

[196] F. Xia, F. Ding, J. Li, X. Kong, L. T. Yang, and J. Ma. Phone2cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing. *Information System Frontiers*, 16(1):95–111, 2014.

[197] 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.

[198] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky. Key ingredients in an iot recipe: Fog computing, cloud computing, and more fog computing. In *Proceedings of the International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE, 2014.

[199] I. Yaqoob, E. Ahmed, A. Gani, S. Mokhtar, M. Imran, and S. Guizani. Mobile ad hoc cloud: A survey. *Wireless Communications and Mobile Computing*, 16(16):2572–2589, 2016.

[200] S. Yi, Z. Hao, Z. Qin, and Q. Li. Fog computing: Platform and applications. In *Proceedings of the Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE, 2015.

[201] S. Yi, C. Li, and Q. Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the Workshop on Mobile Big Data*. ACM, 2015.

[202] S. Yi, Z. Qin, and Q. Li. Security and privacy issues of fog computing: A survey. In *Proceedings of the International conference on wireless algorithms, systems, and applications*. Springer, 2015.

[203] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record*, 34(3):44–49, 2005.

[204] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the Conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[205] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[206] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. USENIX, 2008.

[207] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

[208] W. Zhang, Y. Wen, J. Wu, and H. Li. Toward a unified elastic computing platform for smartphones with cloud support. *IEEE Network*, 27(5):34–40, 2013.

[209] Y. Zhang, H. Liu, L. Jiao, and X. Fu. To offload or not to offload: an efficient code partition algorithm for mobile cloud computing. In *Proceedings of the International Conference on Cloud Networking (CLOUDNET)*. IEEE, 2012.

Appendix

A. Tasklet Language

⟨*program*⟩ ::= ⟨*block*⟩ .

⟨*block*⟩ ::= { ⟨*constDef*⟩ } { ⟨*varDekl*⟩ } { ‘`PROCEDURE`’ ⟨*datatype*⟩ ⟨*ident*⟩ ⟨*parameterList*⟩
‘`{`’ ⟨*block*⟩ ‘`}`’ } ‘`{`’ statement ‘`}`’ .

⟨*statement*⟩ ::= { (⟨*simpleStatement*⟩ ‘`;`’)
| ⟨*conditionalStatement*⟩
| ⟨*loopStatement*⟩ } .

⟨*simpleStatement*⟩ ::= ⟨*ident*⟩ [⟨*index*⟩] ‘`:=`’ ⟨*expression*⟩ | ⟨*procedureCall*⟩
| ⟨*taskletInput*⟩ | ⟨*taskletOutput*⟩
| ⟨*ident*⟩ ‘`:=`’ ⟨*assignArray*⟩ | ⟨*ident*⟩ ‘`:=`’ ⟨*string*⟩
| ‘`RETURN`’ ⟨*expression*⟩ | ⟨*ident*⟩ ⟨*incrOrDecr*⟩
| ⟨*ident*⟩ ⟨*arrayCopy*⟩ ⟨*ident*⟩ .

⟨*conditionalStatement*⟩ ::= ‘`IF`’ ‘`(`’ ⟨*condition*⟩ ‘`)`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’ [‘`ELSE`’ ‘`{`’ ⟨*statement*⟩
‘`}`’] .

⟨*loopStatement*⟩ ::= ‘`WHILE`’ ‘`(`’ ⟨*condition*⟩ ‘`)`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’ .

⟨*condition*⟩ ::= ‘`!`’ ⟨*expression*⟩
| ⟨*expression*⟩ ⟨*relationalOp*⟩ ⟨*expression*⟩ .

⟨*stdFunc*⟩ ::= ‘`length`’ ‘`(`’ ⟨*ident*⟩ ‘`)`’
| ‘`random`’ ‘`(`’ ⟨*expression*⟩ | ⟨*datatype*⟩ ‘`)`’
| (‘`nroot`’ | ‘`pow`’) ‘`(`’ ⟨*expression*⟩ ‘`,`’ ⟨*expression*⟩ ‘`)`’
| (‘`sqrt`’ | ‘`sin`’ | ‘`cos`’ | ‘`tan`’ | ‘`log`’ | ‘`log2`’ | ‘`log10`’ | ‘`exp`’) ‘`(`’ ⟨*expression*⟩ ‘`)`’
.

⟨*keyword*⟩ ::= (‘`rangeLowerBound`’ | ‘`rangeUpperBound`’
| ‘`rangeLowerBoundF`’ | ‘`rangeUpperBoundF`’) .

⟨*varDekl*⟩ ::= ⟨*datatype*⟩ [⟨*index*⟩] ⟨*ident*⟩ { ‘`,`’ ⟨*ident*⟩ } ‘`;`’ .

⟨*constDef*⟩ ::= ‘`CONST`’ ⟨*datatype*⟩ ⟨*ident*⟩ ‘`:=`’ ⟨*value*⟩ { ‘`,`’ ⟨*ident*⟩ ‘`:=`’ ⟨*value*⟩ } ‘`;`’ .

⟨procedureCall⟩ ::= ⟨ident⟩ ⟨parameters⟩ .

⟨taskletInput⟩ ::= ‘>>’ ⟨ident⟩ .

⟨taskletOutput⟩ ::= ‘<<’ ⟨expression⟩ .

⟨expression⟩ ::= [⟨plusOrMinus⟩] ⟨term⟩ { ⟨plusOrMinus⟩ ⟨term⟩ } .

⟨term⟩ ::= ⟨factor⟩ { (‘*’ | ‘/’ | ‘%’) ⟨factor⟩ } .

⟨factor⟩ ::= ⟨ident⟩ [⟨index⟩]
| ⟨value⟩
| ‘(’ ⟨expression⟩ ‘)’
| ⟨procedureCall⟩
| ⟨stdFunc⟩
| ⟨keyword⟩ .

⟨index⟩ ::= ‘[’ ⟨expression⟩ ‘]’ .

⟨arrayCopy⟩ ::= ‘->’ .

⟨parameterList⟩ ::= ‘(’ [⟨datatype⟩ ⟨ident⟩ { ‘,’ ⟨datatype⟩ ⟨ident⟩ }] ‘)’ .

⟨parameters⟩ ::= ‘(’ [⟨value⟩ { ‘,’ ⟨value⟩ }] ‘)’ .

⟨relationalOp⟩ ::= (‘=’ | ‘#’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’) .

⟨plusOrMinus⟩ ::= (‘+’ | ‘-’) .

⟨incrOrDecr⟩ ::= (‘++’ | ‘--’) .

⟨string⟩ ::= ‘"’ { ⟨digit⟩ | ⟨character⟩ | ⟨symbol⟩ } ‘"’ .

⟨assignArray⟩ ::= ‘{’ ⟨value⟩ {‘,’ ⟨value⟩ } ‘}’ .

⟨ident⟩ ::= { ⟨character⟩ } .

⟨symbol⟩ ::= (‘.’ | ‘:’ | ‘,’ | ‘-’ | ‘_’) .

⟨value⟩ ::= (‘’’ ⟨character⟩ ‘’’ | ⟨number⟩ | ⟨boolean⟩) .

⟨datatype⟩ ::= (‘INT’ | ‘FLOAT’ | ‘CHAR’ | ‘VOID’ | ‘BOOL’) .

⟨boolean⟩ ::= (‘TRUE’ | ‘FALSE’) .

⟨number⟩ ::= (⟨integer⟩ | ⟨float⟩) .

⟨float⟩ ::= ⟨integer⟩ ‘.’ ⟨digit⟩ { ⟨digit⟩ } .

⟨integer⟩ ::= ‘1...9’ { ⟨digit⟩ } | ‘0’ .

⟨character⟩ ::= (‘A...Z’ | ‘a...z’) .

⟨digit⟩ ::= ‘0...9’ .

B. Tasklet Protocol Overview

This section gives an overview of the protocol messages. The entities in the system are: the consumer application (APP), the consumer middleware (CM), the broker (B), the provider middleware (PM), and the provider TVM (PTVM). In case of a local execution, the consumer acts also as provider. The overview can be found in Table B.1. The messages for the visualization of the Tasklet execution are omitted. The first four messages, starting with the prefix *i*, are the interface messages between the application and the middleware on the consumer side. The seven broker messages realize the broker communication. They have the prefix *b*. All Tasklet messages have the prefix *t* and the management message start with an *m*.

Type	**Sender**	**Receiver**	**Description**
iRequestMessage	APP	CM	plain Tasklet request
iResendRequestMessage	APP	CM	reparameterization based on previous Tasklet
iResultMessage	CM	APP	return of Tasklet result
iCodeDebugMessage	CM	APP	debug information for Tasklet developers
bIPMessage	B	CM/PM	initial reply of broker
bHeartbeatMessage	CM/PM	B	heartbeat to the broker
bBenchmarkMessage	PM	B	benchmark information
bRequestMessage	CM	B	request for providers
bResponseMessage	B	CM	resource response
bVmUpMessage	PM	B	idle TVM message
bVmDownMessage	PM	B	busy TVM message
tForwardMessage	CM	PM	forwarding Tasklet to assigned PM
tExecuteMessage	PM	PTVM	Tasklet transferred for execution
tResultMessage	PTVM/PM	PM/CM	result forwarded to PM or CM
tSnapshotMessage	PTVM/PM	PM/CM	snapshot message to CM
tHeartBeatMessage	PM	CM	reliable Tasklet execution
mTvmJoinMessage	PM	PTVM	registers TVM at TVMM
mTvmRequestStatusMessage	PM	PTVM	request execution information
mTvmStatusMessage	PTVM	PM	execution information exchange
mTvmPauseMessage	PM	PTVM	pauses a Tasklet execution
mTvmContinueMessage	PM	PTVM	a paused execution is resumed
mTvmSnapshotStopMessage	PM	PTVM	forces snapshot and cancellation
mTvmCancelMessage	PM	PTVM	cancels current Tasklet
mTvmTerminationMessage	PM	PTVM	terminates TVM

Table B.1.: Overview of the Tasklet protocol messages. Messages for visualization of Tasklet execution are omitted.

C. Example Code

C.1. Option Pricing

C--Code

```
float S, dt, sigma,K, r, T, value, nSimulationsF, result;
int nSimulations, call, nSteps;


procedure float uniformInterval(float a, float b){
   return a + random(1.0) * (b-a);
}

procedure float marsagliaPolar(){
   float d, x, y, n;
   x:=uniformInterval(-1.0, 1.0);
   y:=uniformInterval(-1.0, 1.0);
   d:=x*x + y*y;
   n:=d;
   while (n >= 0.0){
      if(n < 1.0){
         if( n > 0.0){
            n := -1.0;
         }
      }
      if(n > 0.0){
         x:=uniformInterval(-1.0, 1.0);
         y:=uniformInterval(-1.0, 1.0);
         d:=x*x + y*y;
         n:=d;
      }
   }
   return x * sqrt(-2.0 * log(d) / d);
}

procedure float optionPricing(){
   float price, value;
   int i,j;
   price := 0.0;
   value := 0.0;
   i := 0;
   while(i < nSimulations){
      price := S;
      j := 0;
      while(j < nSteps){
         price := price + r * price * dt + sigma * price * sqrt(dt) *
             marsagliaPolar();
         j++;
      }
      if(call=1){
         if((price - K) > 0.0){
            price := price - K;
         }else{
            price := 0.0;
         }
      }else{
         if((K-price) >= 0.0){
            price := K - price;
         }else{
            price := 0.0;
         }
      }
      value := value + price;
      i++;
   }
   return ((value / nSimulationsF) * exp(-r * T));
}

>>S;
>>K;
>>r;
>>sigma;
>>dt;
>>T;
>>call;
>>nSteps;
nSimulations := rangeUpperBound;
nSimulationsF := rangeUpperBoundF;

result:=optionPricing();
<<result;
```

Java Code

```
public float[] europeanOption(float stockPrice, float strikePrice, float
    volatility, float interestRate,
         float maturity, boolean isCall) {

      float dt = (float) (maturity * 1.0 / (nTimeSteps - 1));
      float nSimulationsF = nSimulations;
      TaskletBundle taskletBundle = TaskletBundle.fromFile("optionPricing.cmm");

      taskletBundle.addFloat("S", stockPrice);
      taskletBundle.addFloat("K", strikePrice);
      taskletBundle.addFloat("r", interestRate);
      taskletBundle.addFloat("sigma", volatility);
      taskletBundle.addFloat("dt", dt);
      taskletBundle.addFloat("T", maturity);
```

```
        if (isCall) {
            parameterList.addInt("call", 1);
        } else {
            parameterList.addInt("call", 0);
        }

        taskletBundle.addInt("nSteps", nTimeSteps);



        if(timerActivated){
            taskletBundle.setTimeout(maxTime);
        }
        else{
            taskletBundle.setTimeout(2000000L);
        }
        taskletBundle.getQoCList().setReliable();
        taskletBundle.getQoCList().setMigration(true, true, 0);
        taskletBundle.getQoCList().setPartitioning(0, nSimulationsF, 1, true,
             false);
        taskletBundle.start();

        TaskletResultPool allResults = taskletBundle.waitForAllResults();


        return allResults;
    }
```

C.2. Mandelbrot Set

C-- Code

```
float rReal, rImg, startReal, startImg, xShift, yShift, hDiff, vDiff, zoom,
    width, height, startHeight, endHeight, i, j;
int maxIterations;

procedure float abs(float aReal, float aImg){

   return sqrt(aReal * aReal + aImg * aImg);
}

procedure void mul(float aReal, float aImg, float bReal, float bImg){

   rReal := (aReal * bReal) - (aImg * bImg);
   rImg := (aReal * bImg) + (aImg * bReal);

}

procedure void square(float aReal, float aImg){
   mul(aReal, aImg, aReal, aImg);
}

procedure void sub(float aReal, float aImg, float bReal, float bImg){
   rReal := aReal - bReal;
   rImg := aImg - bImg;
}

procedure void add(float aReal, float aImg, float bReal, float bImg){
   rReal := aReal + bReal;
   rImg := aImg + bImg;
}

procedure float iterate(float zReal, float zImg){
   square(zReal, zImg);
   zReal := rReal;
   zImg := rImg;
   add(zReal, zImg, startReal, startImg);
   return abs(rReal, rImg);
}

procedure int main(){
   int counter;
   float result;
   counter := 1;
   rReal:= 0.0;
   rImg:= 0.0;
   result := iterate(rReal,rImg);
   while(result < 2.0){
      if(counter>maxIterations){
         return counter;
      }
      result := iterate(rReal,rImg);
      counter++;
   }
   return counter;
}


>>xShift; >>yShift;
>>width; >>height;
>>zoom;
>>maxIterations;

startHeight := rangeLowerBound;
endHeight := rangeUpperBound;


hDiff := 3.0/(width-1.0);
hDiff := hDiff / zoom;
vDiff := 2.0/(height-1.0);
vDiff := vDiff / zoom;


i := startHeight;
while(i < endHeight+1.0){
   startImg := 1.0 - yShift - (i * vDiff);
   j := 0.0;
   while(j < width){
      startReal := -2.0 + xShift + (j * hDiff);
      <<main();
      j := j+1.0;
   }
   i := i + 1.0;
}
```

Java Code

```
public BufferedImage computeCurrentImage(int width, int height, ZoomParameter
    currentZoomParameters) {

      int startHeight = 0;
      int endHeight = 0;
      TaskletBundle taskletBundle = TaskletBundle.fromFile("mandelbrot.cmm");

      taskletBundle.addFloat("xShift", (float) currentZoomParameters.getxShift()
          );
    taskletBundle.addFloat("yShift",(float) currentZoomParameters.getyShift())
        ;
    taskletBundle.addFloat("width",(float) width);
    taskletBundle.addFloat("height",(float) height);
    taskletBundle.addFloat("zoom",(float) currentZoomParameters.getZoom());
    taskletBundle.addInt("maxIterations",GlobalParameters.ITERATIONS);

    taskletBundle.getQoCList().setReliable();;
    taskletBundle.getQoCList().setMigration(true, true, 0);
    taskletBundle.getQoCList().setPartitioning(startHeight, endHeight, 1, true
        , true);

    taskletBundle.setTimeout(200000L);
    taskletBundle.start();

    TaskletResultPool allResults = taskletBundle.waitForAllResults();


    // iterate over every tasklet
    int color = 0;
    int j = currentTaskletIndex;
    for (TaskletResult result : allResults.values()) {
       System.out.println("Getting results for tasklet: " + j);

       // get all RGB values for the pixels
       for (int i = 0; i < result.size(); i++) {
          color = result.getInt(i);
          if (color < GlobalParameters.ITERATIONS) {
             color %= 255;
             color++;
             if(i / width + (j - currentTaskletIndex) * numberOfRowsPerTasklet
                  < height){
                canvas.setRGB(i % width, i / width + (j - currentTaskletIndex)
                     * numberOfRowsPerTasklet,
                      (new Color(color, Math.abs(color - 120), 255 - color).
                          getRGB()));
             }
          }
       }
       j++;
    }
    this.currentZoomParameters = currentZoomParameters;
    widthOfTheCurrentImage = width;
    heightOfTheCurrentImage = height;
    return canvas;
 }
```

C.3. Prime Number Finder

C-- Code

```
int low,high,result;

procedure  int  checkprime  (int  a){
   int  c;
   c:=2;
   while(c<=(a-1)){
      if((a%c)=0){
         return  0;
      }
      c:=c+1;
   }
   if(c=a){
   return  a;
   }
}



low:=rangeLowerBound;

high:=rangeUpperBound;


while(low<high){
   result:=checkprime(low);
   if(result  #  0){
      <<result;
   }
   low:=low+1;
}
```

Java Code

```
public int[] computePrimes(int low, int high) {
      TaskletBundle taskletBundle = TaskletBundle.fromFile("primesPartitioning.
          cmm");


      taskletBundle.getQoCList().setReliable();
      taskletBundle.getQoCList().setMigration(true, true, 0);
      taskletBundle.getQoCList().setPartitioning(low, high, 1, true, true);
```

```
        taskletBundle.start();
        TaskletResultPool allResults = taskletBundle.waitForAllResults();
        return allResults;

    }
```

Publications Contained in this Thesis

- Martin Breitbach, Dominik Schäfer, Janick Edinger, and Christian Becker: Context-Aware Data and Task Placement in Edge Computing Environments. In: Proceedings of the 2019 IEEE International Conference on Pervasive Computing and Comminications (PerCom 2019), Kyoto, Japan, March 2019.
- Janick Edinger, Laura Marie Edinger-Schons, Aleksander Stelmaszczyk, Dominik Schäfer, and Christian Becker: Of Money and Morals - The Contingent Effect of Monetary Incentives in Peer-to-Peer Volunteer Computing. In: Proceedings of the 52th Annual Hawaii International Conference on System Sciences, 2019 (HICSS 2019), Hawaii, USA, January 2019.
- Dominik Schäfer, Janick Edinger, Martin Breitbach, and Christian Becker: Workload Partitioning and Task Migration to Reduce Response Times in Heterogeneous Computing Environments. In: Proceedings of the 27th IEEE International Conference on Computer Communication and Networks (ICCCN 2018), Hangzhou, China, August 2018.
- Dominik Schäfer, Janick Edinger, and Christian Becker: GPU-Accelerated Task Execution in Heterogeneous Edge Environments. In: Proceedings of the 1st International Workshop on Edge Computing and Networking (ECN 2018) in conjunction with the 27th IEEE International Conference on Computer Communication and Networks (ICCCN 2018), Hangzhou, China, August 2018.
- Melanie Heck, Janick Edinger, Dominik Schäfer, and Christian Becker: IoT Applications in Fog and Edge Computing: Where are We and Where are We Going?. In: Proceedings of the 1st International Workshop on Edge Computing and Networking (ECN 2018) in conjunction with the 27th IEEE International Conference on Computer Communication and Networks (ICCCN 2018), Hangzhou, China, August 2018.

- Dominik Schäfer, Janick Edinger, Jens Eckrich, Martin Breitbach, and Christian Becker: Hybrid Task Scheduling for Mobile Devices in Edge and Cloud Environments. In: Proceedings of the Second International Workshop on Smart Edge Computing and Networking (SmartEdge 2018) in conjunction with the IEEE International Conference on Pervasive Computing and Communications Demonstrations (PerCom), Athens, Greece, March, 2018

- Sunyanan Choochotkaew, Hirozumi Yamaguchi, Teruo Higashino, Dominik Schäfer, Janick Edinger, and Christian Becker: Self-adaptive Resource Allocation for Continuous Task Offloading in Pervasive Computing. In: Proceedings of the International Workshop on Pervasive Flow of Things (PerFoT 2018) in conjunction with the IEEE International Conference on Pervasive Computing and Communications Demonstrations (PerCom), Athens, Greece, March, 2018

- Dominik Schäfer, Janick Edinger, Tobias Borlinghaus, Justin Mazzola Paluska, and Christian Becker: Using Quality of Computation to Enhance Quality of Service in Mobile Computing Systems. In: Proceedings of the 25th ACM International Symposium on Quality of Service (IWQoS), Vilanova, Spanien, June 2017.

- Janick Edinger, Dominik Schäfer, Christian Krupitzer, Vaskar Raychoudhury, and Christian Becker: Fault-Avoidance Strategies for Context-Aware Schedulers in Pervasive Computing Systems. In: Proceedings of the 2017 IEEE International Conference on Pervasive Computing and Comminications. (PerCom 2017), Hawaii, USA, March 2017.

- Janick Edinger, Dominik Schäfer, Christian Becker: Developing Distributed Computing Applications with Tasklets. In: Proceedings of the 2017 IEEE International Conference on Pervasive Computing and Communications Demonstrations (PerCom Demos 2017), Hawaii, USA, March 2017.

- Dominik Schäfer, Janick Edinger, Martin Breitbach, and Christian Becker: Writing a Distributed Computing Application in 7 Minutes with Tasklets. In: Proceedings of the Posters and Demos Session of the 17th International Middleware Conference (Middleware 2016), Trento, Italy, December 2016.

- Janick Edinger, Dominik Schäfer, and Christian Becker: Decentralized Scheduling for Tasklets. In: Proceedings of the Posters and Demos Session of the 17th International Middleware Conference (Middleware 2016), Trento, Italy, December 2016.
- Dominik Schäfer, Janick Edinger, Sebastian VanSyckel, Justin Mazzola Paluska, and Christian Becker: Tasklets: "Better than Best-Effort" Computing. In: Proceedings of the 25th IEEE International Conference on Computer Communication and Networks (ICCCN 2016), Hawaii, USA, August 2016.
- Dominik Schäfer, Janick Edinger, Sebastian VanSyckel, Justin Mazzola Paluska, and Christian Becker: Tasklets: Overcoming Heterogeneity in Distributed Computing Systems. In: Proceedings of the 1st Workshop on Edge Computing (WEC 2016) in conjunction with the 36th IEEE International Conference on Distributed Computing Systems (ICDCS 2016), Nara, Japan, June 2016.

Lebenslauf

Seit 02/2013	Akademischer Mitarbeiter Lehrstuhl für Wirtschaftsinformatik II Universität Mannheim
08/2010 – 01/2013	Master of Science Wirtschaftsinformatik Universität Mannheim
08/2007 – 08/2010	Bachelor of Science Wirtschaftsinformatik Universität Mannheim