UNIVERSITÄT MANNHEIM

# Context-Aware Task Scheduling in Distributed Computing Systems

Inauguraldissertation

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

vorgelegt von

Janick Edinger

aus Mannheim

Prodekan: Prof. Dr. Moritz Fleischmann
Referent: Prof. Dr. Christian Becker
Korreferent: Prof. Dr. Matti Rossi

Tag der mündlichen Prüfung: 29. April 2019

Prüfungsausschuss:
Prof. Dr. Christian Becker (Vorsitzender)
Prof. Dr. Hartmut Höhle

*For my family, who taught me trust.*
*For Laura, who taught me love.*
*For Lion, who taught me priorities.*
*For Dominik, who taught me that 4a.m.*
*is the perfect time for new research ideas.*

# Abstract

These days, the popularity of technologies such as machine learning, augmented reality, and big data analytics is growing dramatically. This leads to a higher demand of computational power not only for IT professionals but also for ordinary device users who benefit from new applications. At the same time, the computational performance of end-user devices increases to meet the demands of these resource-hungry applications. As a result, there is a coexistence of a huge demand of computational power on the one side and a large pool of computational resources on the other side. Bringing these two sides together is the idea of computational resource sharing systems which allow applications to forward computationally intensive workload to remote resources. This technique is often used in cloud computing where customers can rent computational power. However, we argue that not only cloud resources can be used as offloading targets. Rather, idle CPU cycles from end-user administered devices at the edge of the network can be spontaneously leveraged as well. Edge devices, however, are not only heterogeneous in their hardware and software capabilities, they also do not provide any guarantees in terms of reliability or performance. Does it mean that either the applications that require further guarantees or the unpredictable resources need to be excluded from such a sharing system?

In this thesis, we propose a solution to this problem by introducing the Tasklet system, our approach for a computational resource sharing system. The Tasklet system supports computation offloading to arbitrary types of devices, including stable cloud instances as well as unpredictable end-user owned edge resources. Therefore, the Tasklet system is structured into multiple layers. The lowest layer is a best-effort resource sharing system which provides lightweight task scheduling and execution. Here, best-effort means that in case of a failure, the task execution is dropped and that tasks are allocated to resources randomly. To provide execution guarantees such as a reliable or timely execution, we add a Quality of Computation (QoC) layer on top of the best-effort execution layer. The QoC layer enforces the guarantees for applications by using a context-aware task scheduler which monitors the available resources in the computing environment and performs the matchmaking between resources and tasks based on the current state of the system. As edge resources are controlled by individuals, we consider the fact that these users need to be able to decide with whom they want to share their resources and for which price. Thus, we add a social layer on top of the system that allows users to establish friendship connections which can then be leveraged for social-aware task allocation and accounting of shared computation.

## Acknowledgments

This thesis would not have been possible without the support of many people. Here, I would like to take the chance and thank all these people who have supported me throughout the past six years in so many different ways.

First, I would like to thank Prof. Dr. Christian Becker for the guidance and support during my time at the chair of Information Systems II. Christian, listing all the anecdotes from the past years here, would easily fill another 200 pages. Instead, I simply want to say thank you for being a teacher when I needed to catch up in computer science, for being a supervisor when I needed advice for a paper, for being a counselor when I needed to make work or life decisions, and for being a friend when we traveled Boston, Budapest, Hong Kong, Hawaii, Athens, and Kyoto. You always managed to find the fine line between giving me guidance when necessary and providing me the opportunity to follow my own ideas. I am thankful for all of this.

I would like to thank Prof. Dr. Hartmut Höhle for accepting me in his team and for introducing me to enterprise systems research. With every conversation we have, I understand a little more about theoretical contributions. I am looking forward to all the projects that we have already planned together.

I would like to thank Prof. Dr. Matti Rossi, who immediately agreed to be a reviewer for this thesis. It would be an honor for me to work together with you in future research projects.

During my time as a (PhD) student there were multiple people who I consider as mentors as they tought me so much about life in general, and in particular, research and teaching. First of all, there is Prof. Dr. Gregor Schiele, who already believed in me when I was still struggling to write my first lines of code in Java as a student. Gregor, thank you for always taking time when I needed your help or advice. I am grateful for the support of Dr. Justin Mazzola Paluska who mentored us in the very beginning of the Tasklet project. Our meetings at MIT are some of the most memorable experiences during my time as a PhD student. Dr. Sebastian VanSyckel, or Seb for short, thank you for taking the lead in my first Tasklet paper and for supporting so many of our papers after this. Finally, thank you Philipp (Schaber) for your endless amount of ideas and your critical assessment of my papers. Especially, when I asked for it.

Thank you Dr. Miriam Spering, Prof. Dr. Dinesh Pai, and Prof. Dr. Jiannong Cao for hosting me during my stays at the University of British Columbia and the Hong Kong Polytechnic University. Miriam, thank you for introducing me to the fascinating world of visual sciences and for keeping up this collaboration and friendship over so many years until today.

Many thanks to the best team I can imagine that provided me with a pleasant and inspiring working environment. Especially in the most stressful and demanding times in which we spent day and night working towards deadlines, the level of helpfulness and team spirit was beyond words. Thank you to Dr. Patricia Arias-Cabarcos, Martin Breitbach, Kerstin Goldner, Melanie Heck, Benedikt Kirpes, Sonja Klingert, Dr. Christian Krupitzer, Markus Latz, Jens Naber, Yugo Nakamura, Martin Pfannemüller, Dr. Vaskar Raychoudhury, Dr. Felix Maximilian Roth, Dr. Dominik Schäfer, and Anton Wachner. In particular, I want to thank Dominik, for all the time that we have spent working together starting from the very early days as student assistants. Your dedication to the project in endless night shifts, your optimism in situations that seemed hopeless ('*What should go wrong?*'), and your friendship made you the ideal project partner. I will miss our discussions. Pitzi, thank you for all the years of managing us and for joining multiple night shifts to get the most out of every paper. I admire your discipline and loyalty. Thank you Max and Jens for lightening my workload especially during the last year which allowed me to work on this thesis. Pfanni, thank you for always being there when help is needed and for your continuous improvements for our team (Git server, exam templates, registration tool, CT2 lectures, ...). To the Breitis: Thank you for all your support over the years as student assistant, co-author, and colleague and for continuing the Tasklet project in the future. Thank you, Anton, as working with you in the EyeT Lab project is always exciting and a great experience. Also, many thanks to you, Kerstin, for all your support in administrative tasks throughout the years.

Thank you to the team of the chair of Enterprise Systems for the warm welcome and for sharing your research ideas with me. This includes Karoline Glaser, Brigitte Haber, Tobias Nisius, Florian Pethig, and Fareed Zandkarimi.

Thank you to my co-authors from other groups and universities. The joint work across the boundaries of departments and countries makes research much more engaging. Thank you Prof. Dr. Teruo Higashino, Prof. Dr. Andy Schürr, Prof. Dr. Heiner Stuckenschmidt, Dr. Hirozumi Yamaguchi, Tim Bauer, Bastian Biemer, Sunyanan Choochotkaew, Alexandra Hofmann, Robin Klose, Roland Kluge, Manisha Luthra, Timo Sztyler, and Markus Weckesser. This work was supported by the German Research Foundation (DFG) under grant BE 2498/10−1 'Tasklets: Ein Ansatz für Best-Effort Computing'. Thanks to the students who supported the Tasklet project within their theses and beyond. They are in particular: Erik Baumert, Tobias Borlinghaus, Tianhong Chen, Simona Doneva, Lukas Draheim, Anna Florentine Eitelbach, Daniel Flachs, Niklas Frühauf, Philipp Kaiser, Martin Schaumann, Nadja Seemann, and Aleksander Stelmaszczyk.

I would like to thank my family who has provided me with unconditional support in every stage of my life. At all times, you never hesitated to help out even though you had to cancel or change your own plans. To my parents, thank you for always being there for me and for providing me with everything that I needed to pursue my own way. To my sister Nina, thank you for teaching me how to read when we were children. This has turned out to be very useful, indeed. Thanks also to Helga, Wolfgang, and Lisa for babysitting, cooking, and housekeeping so many times during the final stage of this thesis. Thank you, Sunny, for taking me out for a walk when the sessions on the computer were getting too long.

Finally, thank you Laura for being all of the above. You are my co-author, my mentor, my wife, and my best friend. Thank you for following me around the world and for taking me to new places. Thank you for bearing with me in these intensive last months of this thesis. Thank you for this wonderful miracle that has entered our life more than a year ago and for taking care of this miracle when I needed to work on this thesis. I cannot imagine a life without you anymore.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **AOP** | Aspect-Oriented Programming |
| **API** | Application Programming Interface |
| **BOINC** | Berkeley Open Infrastructure for Network Computing |
| **CAN** | Content Addressable Network |
| **CPU** | Central Processing Unit |
| **CSMA** | Carrier Sense Multiple Access |
| **DHT** | Distributed Hash Table |
| **EC2** | Elastic Compute Cloud |
| **ETSI** | Europäisches Institut für Telekommunikationsnormen |
| **FIFO** | First In - First Out |
| **FPGA** | Field Programmable Gate Array |
| **GAE** | Google App Engine |
| **GHz** | Gigahertz |
| **GPU** | Graphics Processing Unit |
| **HTTP** | Hypertext Transfer Protocol |
| **IoT** | Internet of Things |
| **IP** | Internet Protocol |
| **ISP** | Internet Service Provider |
| **IT** | Information Technology |
| **MBS** | Mandelbrot Set |
| **MHz** | Megahertz |
| **OS** | Operating System |
| **P2P** | Peer-to-Peer |
| **PDF** | Probability Density Function |
| **QoC** | Quality of Computation |
| **QoS** | Quality of Service |
| **RPC** | Remote Procedure Call |
| **RTP** | Real-Time Transport Protocol |
| **SLA** | Service Level Agreement |
| **SMTP** | Simple Mail Transfer Protocol |
| **SOA** | Service-oriented Architecture |
| **TCP** | Transmission Control Protocol |
| **TVM** | Tasklet Virtual Machine |
| **UDP** | User Datagram Protocol |
| **WAN** | Wide Area Network |
| **WWW** | World Wide Web |

# 1. Introduction

For computer users who frequently run computationally intensive applications, processing power is inherently a scarce resource. Some software tools, such as OMNeT++[1], provide a straightforward solution to split up the workload into individual packages and to distribute these work packages across the available CPU cores of the local device. However, this leveraging of existing resources is limited to the processing units of the single device when, at the same time, devices in proximity might run idle. The additional computation power of these devices could speed up the execution of the resource-hungry task substantially. Even within the local computer, there might be other processing units such as a GPU that could help to accelerate the execution of the task. However, there is no straightforward way to make use of all these available resources. This is even more surprising as, in the literature, there are various approaches that have demonstrated the technical feasibility of sharing computational workload among distributed devices[2].

In a time in which we share basically everything, including our private stories[3], car rides[4], bicycles[5], and even apartments[6], how come that sharing of computational resources has never made it beyond a niche for tech-savvy geeks who are interested in finding extraterrestrial life [1]? Why is the plethora of existing resources, which are technically only milliseconds away from our own devices, practically unreachable for our local applications?

In this thesis, we want to - literally - think outside the box and present an abstraction for computation that allows to distribute workload beyond the boundaries of our local devices. In our vision, each computational device can execute tasks

[1]OMNeT: https://aws.amazon.com/de/ec2/, accessed: 20/03/2019
[2]For a comprehensive survey about these approaches, please refer to Chapter 3.
[3]Facebook: www.facebook.com, accessed: 20/03/2019
[4]Uber: www.uber.com, accessed: 20/03/2019
[5]Spinlister: www.spinlister.com, accessed: 20/03/2019
[6]Airbnb: www.airbnb.com, accessed: 20/03/2019

for applications on every other device. Computation itself becomes a commodity similar to water or electricity. Devices can exchange computation without barriers caused by heterogeneity of hardware, software, or non-functional requirements of the tasks. Thus, computation becomes independent from the resources of the local device and can easily be exchanged among distributed devices.

This approach has multiple advantages regarding performance, costs, and sustainability. First, resources could be used more efficiently. Instead of deploying dedicated machines for each application individually, resources can be used for multiple applications. This can increase the utilization of the resources. As existing devices can be used more efficiently, fewer hardware has to be produced, deployed, and eventually disposed. This lowers the consumption of already limited resources. Second, as a consequence of the former, the amount of required devices that run at the same time can be reduced. This lowers the overall energy consumption and, thus, does not only save costs but also benefits the environment. Third, applications that can benefit from parallelization can experience significant speed-ups. As the resource pool becomes large, tasks can be split up into many parallel executions that can run at the same time. For a short period of time, the application uses a large number of resources and finishes the execution earlier. Fourth, as all devices can contribute to the resource pool, end user devices can be included. This allows to use nearby user-controlled devices as well as cloud resources at the center of the network. Fifth, as the amount of available resources increases, end user devices do not have to be equipped with powerful, expensive hardware anymore. They can rather become thin clients that make use of remote resources. This does not only save costs but can also extend the battery lifetime of these devices.

In the following, we approach the topic of sharing computational resources across heterogeneous devices from a broader perspective. We provide a problem definition that takes existing approaches for computational resource sharing into account and derive the research questions for this thesis.

## 1.1. Problem Definition

The amount of computational power, which is omnipresent in our environment, is increasing steadily. This trend is favored by two independent factors. First, the amount of computational devices is expected to grow continuously [2]. Second, these devices are getting more powerful in terms of computational resources and battery [3]. While these devices are designed to handle peak demands in computational power, their processing units remain idle most of the time. At the same time, new types of applications emerge that require more computational resources than devices typically offer. These applications include machine learning approaches, augmented reality, image processing, and big data analytics.

Distributed computation systems address this imbalance between idle resources on the one hand and resource-hungry applications on the other hand. They augment the amount of available computational power by leveraging remote resources that otherwise remained unused [4]. This allows users to run complex applications even with less powerful and, thus, less expensive hardware. While distributed computation systems have been around for more than two decades and appear in multiple forms, their application is still inflexible and often limited to a predefined setup of resources that can be used for computation. In remote procedure calls (RPCs), where parts of applications are executed remotely, the resources are well known as they are dedicated to serve as providers for computational power [5]. They need to be set up before a remote execution is possible. With the advent of cloud computing, the administrative process of installing, maintaining, and scaling these resources has been simplified to a point that distributed computing applications can be deployed within only a few minutes [6]. Cloud computing offers powerful and stable resources that can be deployed on demand and billed in a pay-as-you-go manner. However, despite the introduced flexibility, cloud computing services still require a strong coupling between applications and resources. Customers need to set up their accounts and start virtual machine instances.

Code offloading systems eliminate this requirement as they do not only pass data or parameters to functions that have already been installed on remote devices. Rather, the function itself is sent. As a result, the offloaded computation can be forwarded to any resource that is capable of executing the code. This has mainly two advantages. First, it decouples applications and remote resources. Resources do not need to remain reserved for one particular application but can serve multiple applications even at the same time (multi-tenancy) [7]. Second, applications can make use of a previously undefined pool of resources that can be leveraged without further maintenance overhead. This allows to include also end user devices into the resource pool just as it is done in desktop grid [8] and volunteer computing systems [9].

The great amount of flexibility, however, comes along with an increased complexity in resource management and scheduling. While offloading to stable cloud resources is straightforward, end user devices are typically unreliable. They might leave the system at any time, abort task executions, temporarily lose network connectivity, or fluctuate in their performance. They further introduce a large variance in terms of hardware, operating system, availability, and reliability. In addition to the heterogeneity of the resources, the wide range of diverse applications also contributes to the complexity of decentralized offloading systems. These applications are written in different programming languages and issue tasks that do not only differ in their complexity but also in their non-functional requirements. Having heterogeneous resources on the one side and heterogeneous tasks on the other side creates a demand for a holistic solution for a resource sharing system that does not only allow to execute tasks remotely but also takes the different characteristics and requirements into account. More precisely, the system needs to provide quality of service (QoS) support as well as context-aware scheduling.

## 1.2. Research Questions

We acknowledge that the vision discussed above is highly ambitious. In this thesis, we do not claim to fully put this vision into practice. Rather, we use it as guideline for each design and implementation decision that we make throughout the thesis. It further helps to identify the research gap that we will discuss in the following and that motivates our research questions. The overall goal of this thesis is to make computation interchangeable between different kinds of devices and for different kinds of applications. As the computational environment is highly heterogeneous, the main challenge is to enable the interoperability of the devices, operating systems, and software. This requires an abstraction for computation that on the one hand allows developers to express an arbitrary application logic and on the other hand is executable by different types of platforms. The creation, scheduling, and execution of this task introduces an overhead. To allow even resource-poor devices to benefit from the resource sharing system, this overhead must be kept minimal. Therefore, we formulate the first research question as follows.

**Research Question 1:** *What is a lightweight abstraction for computation that allows (i) programmers to define their application logic in the form of tasks, (ii) local devices to distribute these tasks to remote resources, and (iii) heterogeneous remote devices to execute them?*

Integrating different types of resources, including unreliable end user devices, results in the lack of any guarantees regarding the reliability or the speed of the execution. Without any countermeasures, this renders the system unusable for those applications that have non-functional requirements beyond a best-effort task execution. In turn, implementing reliability into the system would introduce additional overhead that interferes with the idea of a lightweight approach and excludes resource-poor devices. To solve this dilemma, the system needs to provide a context-aware task scheduler that adaptively provides non-functional guarantees for some applications while others still profit from the lightweight nature of the system. Hence, we derive the following research question.

**Research Question 2:** *How can a context-aware task scheduler implement non-functional guarantees for some applications into a lightweight distributed resource sharing system without introducing additional overhead for other applications or system resources?*

In the discussion about resource sharing, we must not neglect the fact that devices are owned by persons who decide what the resources of these devices are used for. A technically proper and well-implemented sharing system is of no use if device owners are not willing to share their resources and application users do not trust the system. Thus, the design of a comprehensive resource sharing system takes the requirements and the concerns of its users into account. This leads to our third research question.

**Research Question 3:** *What are the enablers and inhibitors for device owners to participate in a computational resource sharing system and what are the implications for the design of such a system?*

## 1.3. Structure

The remainder of this thesis is structured as follows. Chapter 2 covers fundamental knowledge in code offloading and task scheduling. Chapter 3 presents related approaches in these fields. We identify the requirements of a resource sharing system in Chapter 4. Chapters 5 to 7 constitute the main part of this thesis. In Chapter 5, we discuss the first research question. We introduce the Tasklet middleware, our approach for a computational resource sharing system. We discuss the design and the implementation of the system and perform an evaluation of the prototype. In Chapter 6, we present our context-aware task scheduler in response to the second research question of this thesis. We introduce multiple scheduling strategies that enhance the performance of the task execution in the system. We consider the social aspects of computational resource sharing in Chapter 7 to answer the third research question. We identify incentives and obstacles for devices owners to share their resources and design and implement a social overlay on top of our Tasklet middleware. In Chapter 8, we evaluate our system based on the functional and non-functional requirements identified in Chapter 4. Chapter 9 concludes the work and provides an outlook on open research questions.

# 2. Foundations

This chapter introduces the fundamental concepts that this thesis builds upon. It provides the knowledge that is required to follow the argumentation in this work. First, we discuss the evolution of distributed computing and its different stages over time. Second, we present the concept of task offloading. We introduce computation offloading with the focus on application partitioning and the different aspects of the offloading decision. The final section deals with context-aware systems and self-adaptation. The following discussions will not cover the topics in their entirety, but focus on those aspects that are related to the content of the thesis.

## 2.1. Distributed Computing

Modern communication technologies allow to use computational resources across the boundaries of one single physical device. Rather, geographically distributed resources can be accessed and work together in a distributed system. A very popular definition of a distributed system can be found in [10]:

"*A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.*" [10, p.968]

This definition implies two important characteristics of distributed systems. First, the resources are independent devices that provide heterogeneous hardware and software capabilities. Second, this heterogeneity is transparent for users who can work with the system in the same way as they would work with a single device. This is enabled by the use of middlewares [11], which add a layer between the operating system and applications. Middlewares implement several distribution transparencies [12] to hide the complexity from users and programmers. Distributed systems provide a variety of services to users such as communication, storage, and computation. For this work, distributed computation, or distributed computing, is of particular interest.

The trend of distributed computing has emerged from the field of high-performance computing where dedicated supercomputers where used to solve computationally expensive tasks [13]. By distributing the workload over multiple devices, the need for these rare and expensive machines was eliminated. As the devices became able to communicate with each other, multiple resources together could provide the same amount of resources as individual supercomputers.

Until now, the evolution of distributed computing architectures is emerging. In the following, we sketch this evolution of distributed computing architectures and introduce the most important paradigms.

**Cluster Computing:** Cluster computing has been the first paradigm to exploit parallel computing on multiple machines. Clusters consist of rather simple and homogeneous computers that are connected via a high-speed network [10]. In 1994, *Beowulf*, the first actual implemented cluster computing system, was presented [14]. It consisted of 16 motherboards with each 256 megabytes of random access memory, a 100 megahertz processor, and 500 megabytes of disk storage. The responsibilities within a cluster were split between a master node and multiple compute nodes. The master node provides the interface for users of the system and schedules batch tasks to the compute nodes. The compute nodes themselves cannot be accessed remotely but only perform the computation. More modern cluster computing approaches aim to keep the software stack more lightweight [15]. They deploy a thin middleware on the compute nodes to optimize their performance for parallel execution of tasks. Current computer clusters contain tens to even hundreds of thousands of processors [16].

**Grid Computing:** In the mid-1990s, Foster and Kesselmann coined the term *Grid* as a new type of distributed computing system [17]. The term was used analogously to power grids and indicated that this new technology could provide computation and storage capabilities in a similar way as power grids allow people to use electricity. More specifically, grids aim to "*enable resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations*" [17, p.40].

A widely accepted view on grid architectures defines five layers, which provide interfaces on different levels [18]. The *fabric* layer provides direct access to a logical or physical resource. It offers mechanisms to start and monitor the execution of programs and to request hardware and software characteristics of the resource. The *connectivity* layer allows for user authentication at grid resources and data exchange between multiple resources. The *resource* layer provides protocols to monitor and control individual resources. The functions of this layer directly map to the functions of the fabric layer and provide a direct access to the resources. The *collective* layer manages the access to multiple resources and coordinates the monitoring, scheduling, and load-balancing across these resources. Finally, user applications are implemented on the *application* layer and access the underlying interfaces to eventually make use of the resources on the fabric layer.

Typically, the main users of grid computing are scientific institutions that run large computational projects which exceed their local capacity. Participating in these grids means both, providing local resources as well as gaining access to remote resources [6]. Even though most applications of grid computing systems are applied in academic and scientific projects, there are also commercial grid projects. A detailed classification of grid computing systems can be found in [19].

**Desktop Grids and Volunteer Computing:** Desktop grids make use of idle cycles of end user desktop computers [20]. These devices became more powerful and widely distributed among private users in the late 1990s and early 2000s. The aggregated computational power of these resources even exceeds the capabilities of modern supercomputers [21] and can typically be used at low cost [20]. In classical grid environments, resources are administered in a more centralized way by organizations. In contrast, desktop grid resources are controlled by the owners of end user devices and thus do not provide any guarantees regarding performance or availability. Desktop grid systems include several challenges [22]: (i) *Volatility:* Devices might leave the system at any time, even during the execution of a task. (ii) *Dynamic environment:* The system state, such as load and bandwidth, is continuously changing over time. (iii) *Lack of trust:* As resource are anonymous, there is no trust relationship between resource users and resource providers. (iv) *Failure:* Resources might fail at any point in time. (v) *Heterogeneity:* Resources show a large variance in terms of computational power, availability, as well as hardware and software capabilities. (vi) *Scalability:* Resource management in large decentralized distributed computing environments is complex.

Desktop grids can be implemented on a global scale or limited to single organizations [23]. Even though the technical feasibility of large system has been demonstrated in systems such as HTCondor [24] and BOINC [9], the success of desktop grids is limited to so-called volunteer computing systems where devices owners donate their resources to scientific projects for free. Commercial systems such as Entropia [8] have never achieved a comparable popularity. A comprehensive categorization of desktop grid approaches can be found in [22].

**Cloud Computing:** When cloud computing became popular in the early 2000s, there was a lively discussion whether cloud computing is either a new IT paradigm or just a new name for what was formerly known as grid computing [6,25]. Indeed, cloud computing uses similar concepts as known previously from grid computing architectures. Instead of performing computation on our own devices, we outsource the workload to centrally administered, reliably resources [6]. However, the success of cloud computing is unquestionable and can be attributed to a trend that Kleinrock has discussed in 2003 [26]. To this date, the Internet has already been successful in providing an always running, always accessible, omnipresent communication infrastructure. Kleinrock argues that, despite these achievements, the Internet fails to provide a service that "*weaves itself into the fabric of everyday life*" [27, p.94] as Mark Weiser describes it in his vision paper *'The computer for the 21st Century'* [27]. Cloud computing introduces this invisibility to distributed computing as it offers a service-oriented approach to resource provisioning that makes the usage of these resources transparent to users. This does not only follow Weiser's vision but allows for a much broader user range than typical grid computing systems.

Grossman [28] as well as Armbrust *et al.* [29] define three major differences between cloud and grid computing: scale, pricing, and simplicity. Cloud architectures scale to multiple data centers in an elastic way which eliminates the need for complex resource planning. As the usage is metered in a pay-per-usage fashion, no up-front investments are necessary. Finally, cloud resources can be accessed with minimal effort via easy-to-use interfaces.

The NIST definition of cloud computing defines three types of service models for cloud computing [30]. In *Software as a Service (SaaS)*, consumers use centrally administered applications without having the need for developing, installing, or maintaining them. In *Platform as a Service (PaaS)*, cloud providers offer languages, libraries, services, or tools to develop and deploy applications. In *Infrastructure as a Service (IaaS)*, consumers rent whole physical or virtual devices with full control over their operating systems, storage, and deployed applications. As cloud storage systems have become popular, the term *Data as a Service (DaaS)* is also commonly used [31]. Beside these categories, other service models exist, which has led to the term *Everything as a Service (XaaS)*. A comprehensive survey of cloud computing systems can be found in [32].

A particular form of cloud computing is mobile cloud computing [33]. As mobile devices face other challenges than static devices, research in mobile cloud computing overcomes obstacles such as limited performance, fluctuations in the environment, limited battery lifetime, and reduced storage capabilities [34]. It combines mobile web and cloud computing and provides mobile users with ubiquitous computation capabilities that can be efficiently accessed at anytime and from anywhere [35].

**Edge and Fog Computing:** Cloud computing offers solutions for multiple applications. However, with the rise of applications in the Internet of Things (IoT), the requirements of applications are changing [36][1]. Particularly with regard to latencies, cloud resources do not meet the demands of highly-interactive closed-loop systems that require instant responses [37, 38]. Thus, edge and fog computing paradigms move the computational resources from the core of the Internet towards the edge of the network [39]. As the concepts are rather new, a common understanding of edge and fog computing is missing. While the terms are sometimes used interchangeably [40–42], there are some relevant differences between the two approaches.

Edge computing focuses on the general idea to bring computing and storage resources closer to the edge of the network [43]. This idea has been present for almost a decade, as Satyanarayanan *et al.* introduced the concept of Cloudlets, which are small data centers in local area networks that are only one hop away from user devices [44]. Mobile edge computing builds upon this idea and introduces

[1] [36] is joint work with M. Heck, D. Schäfer, and C. Becker

computing and storage resources at base stations in radio access networks [45]. Being only one hop away from the base stations does not only minimize delay and jitter [46, 47], but also allows to consider real time information about the base stations such as their current load [48]. Further, the context information of the mobile devices can be used [48,49]. The European Telecommunications Standards Institute (ETSI) describes mobile edge computing as the "*convergence of IT and telecommunications networking*" [50, p.4].

Throughout this thesis, we follow the definition of Lopez *et al.* [39] who define edge computing as all distributed computing approaches at the edge of the network. This includes end-user administered devices that are typically not directly connected to the backbone of the Internet including "*desktop PCs, tablets, smart phones, and nano data centers (stable computing devices such as routers or media centers)*" [39, p.38].

Fog computing also depends on resources at the edge of the network [51]. The concept was first introduced by Bonomi *et al.* from Cisco as a platform that provides resources to the user typically, but not exclusively, on the edge of the network [52]. In addition to edge resources, fog computing uses traditional cloud resources as a backup for computations that exceed the capabilities of edge resources [53]. Proposed fog architectures consist of three layers [54]. On the lowest layer are fog nodes which are heterogeneous physical resources. The middle layer provides an abstraction of these fog nodes and allows to monitor and control these resources. The top layer is responsible to orchestrate tasks and resources in these systems. A comprehensive survey of fog computing approaches can be found in [55].

## 2.2. Computation Offloading

Computation offloading describes the process of migrating computation to remote resources. Application areas include computationally intensive processes such as speech recognition, natural language processing, computer vision and graphics, machine learning, augmented reality, planning, and decision-making [44]. In contrast to traditional client-server architectures where clients always delegate computation to servers, offloading systems make the offloading decision case-by-case [56]. Offloading differs from remote procedure calls where dedicated servers

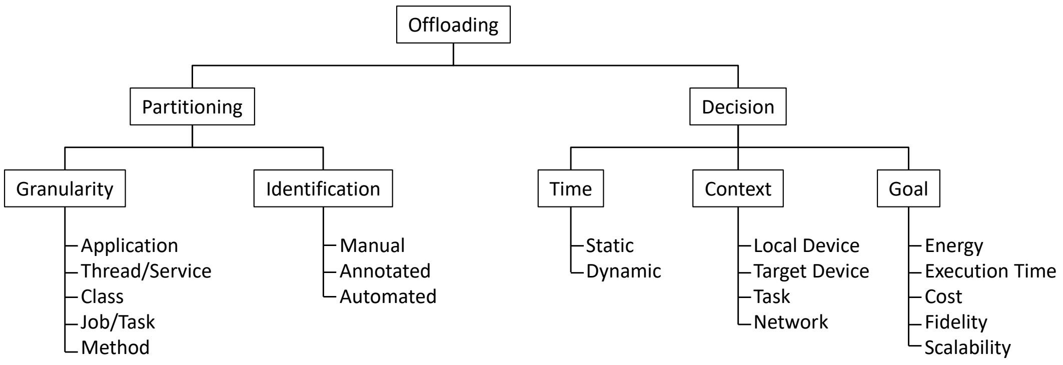


Figure 2.1.: Taxonomy of an computational offloading system. The offloading process consists of the partitioning of the task and the (context-aware) offloading decision.

offer the execution of known procedures [5, 57]. Offloading is also different to cluster and grid computing. Whereas in the latter the idea is to create a large distributed supercomputer, computation offloading aims to provide additional resources for single devices [4]. Typically, these devices are thin clients, which run computationally intensive applications and spontaneously need access to remote resources for a short period of time. In literature, there are two major goals of offloading architectures: improving performance and saving energy. To achieve these goals, offloading systems analyze context parameters such as bandwidth, server speed, available memory, server load, and data exchange to make optimal offloading decisions. In addition, the workload of the computation and the required energy need to be estimated. As these estimations are non-trivial and context parameters change over time, decision making is a complex task that has gained a lot of attention in research. Figure 2.1 illustrates a taxonomy of important issues in computation offloading divided into application partitioning and offloading decision questions. In this section, we discuss how application partitioning works and how offloading decisions are made.

### 2.2.1. Partitioning

Before an application is offloaded, it needs to be identified, which part of the application can be forwarded to remote resources for execution [58]. This process is called application partitioning. Here, we address two issues of application partitioning, namely the level of granularity of the offloaded units as well as the amount of human interaction that is required to perform the partitioning.

**Granularity:** Applications can be offloaded at various granularities. Whereas some frameworks offload the whole application, others provide a fine-granular partitioning which allows for more sophisticated offloading decisions. Many approaches use a remote copy of a virtual machine as a target for offloading [44, 59]. They replicate the local runtime environment on powerful remote resources that can perform the same operations as the local device. At the end of the offloading process, the state of the local device and the virtual remote clone are merged. In application-level offloading, code in form of binaries is sent to remote devices where the whole application is started [8, 60]. Other approaches offload parts of the application on a task-level [9, 61, 62]. This is often the case when the same code is executed multiple times on different data (single instruction, multiple data in Flynn's taxonomy [63]). The code is sent together with the data to the remote machine as a bundle [8, 64, 65]. Fine-grained approaches allow to offload computation on a method or submethod level [58, 66, 67]. The computationally intensive parts of the applications are extracted and offloaded. The result of the remote execution is returned to the local device where the application continues locally.

**Identification:** Not all parts of a program qualify for computation offloading. Some parts are so small that offloading would introduce unnecessary overhead. For others it is not possible to offload them as such. The latter category includes code that implements the user interface, makes use of the local devices sensors, or reads or writes the local state of the application [68].

The identification of the parts that can be offloaded can involve different levels of user interaction. In multiple systems, code for remote execution needs to be written explicitly in a separate method or even in a different programming language [8, 64]. Application developers have to write local and remote code separately. Existing applications have to be rewritten to be used in the offloading framework. Other systems allow to use unmodified source code and add annotations to signal the offloading system which part of the application qualifies for a remote execution [68, 69]. While this approach reduces the workload for developers it is still prone to errors. As a solution, fully automated partitioning schemes identify local and remote parts without developer input [58, 70]. Partitioning does not determine which part of the code eventually gets offloaded. It only identifies the parts of an application that qualify for remote execution.

### 2.2.2. Offloading Decision

The offloading decision is a the central part of each offloading approach. It decides *whether*, *when*, and *what* code gets offloaded. The decision can be taken at development time or during runtime. It can consider multiple parameters of the computation itself, the local device, the remote device, as well as the environment. Further, the decision can optimize multiple execution parameters, such as runtime, battery consumption, or execution cost. Here, we discuss the dimensions *time* and *goal* (compare Figure 2.1). The context dimensions for a context-aware scheduler will be discussed in Section 2.3.

**Time:** Static offloading decisions are taken at development time of the program. As the decision does not take any dynamic parameters into account, it adds only little overhead at runtime. Dynamic parameters, such as server load, bandwidth, and task complexity need to be predicted [71].

Dynamic offloading monitors environmental parameters at runtime and makes the offloading decision just in time [72]. Taking current observations into account allows for a much more informed and thus more optimized decision. At the same time, dynamic algorithms result in more overhead. Monitoring and analyzing context parameters costs both computation cycles and energy. Depending on the use case, the benefits of dynamic decision making can be canceled out by the additional workload.

**Goal:** Offloading decisions typically try to minimize the execution time, the energy consumption, or both [56]. Based on the parameters discussed above, they estimate whether a remote execution is beneficial or not. Inequation 2.1 shows when offloading improves the execution time of a task.

$$\frac{workload}{local\_speed} > \frac{data}{bandwidth} + \frac{workload}{remote\_speed} \tag{2.1}$$

The left side of the inequation represents the local execution time which is determined by the *workload* of the task and the *local processing speed*. The required time increases with a high workload and a low local processing speed. The right side of the equation shows the required time in case of a remote execution which is the sum of the data transfer and the execution time on the remote device. The data transfer increases with the size of the *required data* and a low *bandwidth*.

The remote execution time increases with a high *workload* and a low *remote processing speed*. Offloading is beneficial when the inequation holds true, that is when the local execution time is greater than the remote execution that involves offloading. It can also be noticed that the remote execution time is irrelevant for the decision in cases where the time for the data transfer is already larger than the duration of the local execution.

Besides minimizing the execution time, reducing the battery consumption is the other major goal of computation offloading. When the offloading decision optimizes the energy consumption, Inequation 2.2 is applied.

$$power_l * \frac{workload}{local\_speed} > power_t * \frac{data}{bandwidth} + power_r * \frac{workload}{remote\_speed} \tag{2.2}$$

In the formula, $power_l$, $power_t$, and $power_r$ describe the energy consumption rate of the local processing, the data transfer, and the remote processing respectively. The remote execution is beneficial when the inequation holds true and the local execution requires more energy than the remote execution. The energy consumption of the local execution increases with a high energy consumption rate, a high workload and a low processing speed. The energy consumption of the remote execution increases with a high amount of data, a low bandwidth, high energy demand for data transfer and processing, a high workload, and a low processing speed.

Further goals of offloading systems are cost reduction, application fidelity, and scalability of applications. To reduce costs, tasks can be scheduled to the cheapest resource provider or even make use of free resources [73]. The fidelity of an application is a measure for the quality of the results. Fidelity can have many forms such as the resolution of images, the frame rate for video applications, or the size of the vocabulary in a speech recognition application [74]. Finally, some offloading systems mainly focus on scalability of applications. In the early days of computation offloading, creating a distributed supercomputer to allow new kinds of applications to be executed at low cost was one of the major driver for offloading systems [24, 60].

## 2.3. Context-Aware Scheduling

The second part of this thesis is concerned with context-aware scheduling in distributed computing systems. Resource allocation describes the process of assigning resources to computationally intensive user tasks [75]. In this thesis, we use the words resource allocation and resource scheduling, or simply scheduling, interchangeably. Scheduling consists of three main phases: resource discovery, resource selection, and task execution [76]. First, resource discovery includes finding available resources and maintaining information about these resources in a list [77]. In the second phase, the resource selection, the matchmaking between the task and a resource, happens. Comprehensive surveys about these matchmaking algorithms can be found in [78–80]. Finally, the task is sent to the resource for execution. A successful execution might produce results that are then returned to the initiator of the task.

There is a large number of resource schedulers present in literature. As the nature of distributed computation systems is very diverse, there are different resource allocation approaches required. We limit the scope of this thesis to context-aware scheduling approaches as they can be applied in heterogeneous environments where the quality of the offloading decision depends on the current state of the system. Next, we give a brief introduction to context before we discuss possible dimensions for context-aware scheduling.

### 2.3.1. Context

No computer system is running in complete isolation of any factors that might affect the execution and the performance of this system. In addition, the status of the system itself might have an effect on how it is working. Context-aware applications take these factors into account. Dey and Abowd define context as follows: "*Context is any information that can be used to characterize 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.*" [81, p. 3]. Context-aware applications can monitor the computing environment, adapt their behavior dependent on these measurements, and react to changes of this environment [82].

Context-awareness does not come for free. System developers have to identify relevant context variables and implement a dynamic program flow that adapts to context changes. Thus, context-awareness is typically implemented in self-adaptive systems [83]. In this domain, control structures such as the MAPE cycle [84] are used to gather and process context information. MAPE stands for monitoring context variables, analyzing these measurements, planning adaptations, and eventually executing them. In the following, we will discuss multiple context dimensions that can be used for adaptive, context-aware task allocation strategies.

### 2.3.2. Context-Aware Scheduling Decisions

Whether offloading is beneficial depends on an interplay of multiple parameters. Some of these parameters are known, others can be measured, and some can only be estimated. The processing speed of the devices can be stated by the clock rate of a processor which is static. However, the execution speed of tasks is slowed down when multiple processes are active. Thus, to get reliable estimates about a device's performance, real time information about the current load is required. The available memory on local and remote devices also can be retrieved at runtime before making the offloading decision. For local devices this might be important as memory might be the bottleneck of computationally intensive applications [85]. It also needs to be assured that the remote device has sufficient memory to take over the computation. The connection to the network plays an important role in the offloading decision. Bandwidth, delay, and jitter determine whether code and data can be offloaded efficiently. As these parameters fluctuate, they have to be measured at runtime [86]. The offloading decision depends also on the amount of data that needs to be send along with the task. While the code itself is typically quite small, the required data can be multiple orders of magnitude larger [68]. Finally, the complexity of the task needs to be considered for the offloading decision. For short, highly responsive tasks, the offloading process can add unacceptable overhead. Runtime estimation of tasks, however, is a complex challenge itself and in most cases cannot be performed reliably. There are static and dynamic methods to estimate the execution time. Static approaches analyze the source code and perform control-flow analysis, hardware-level analysis, and the calculation of the worst case execution time. Dynamic methods run parts of the code and base the analysis on these measurements.

# 3. Related Work

This chapter reviews related work of the thesis. The review is divided into two sections, representing the main structure of the thesis. In Section 3.1, we present computation offloading systems that share common research questions with this thesis. Due to the plethora of existing approaches, we only focus on those systems that provide an entire offloading architecture instead of optimizing particular offloading algorithms. For an overview about these approaches, we refer to more in-depth surveys [46, 56, 87]. In Section 3.2, we discuss context-aware scheduling approaches in computation offloading environments. We identify the context dimensions that are considered for the offloading decision and briefly present relevant approaches.

The literature presented here covers related work in offloading systems and context-aware scheduling systems. Further literature analyses about fault-avoidant scheduling, decentralized scheduling, and social scheduling, are embedded into the relevant chapters such as in Sections 6.2, 6.3.1, and 7.1.4.

## 3.1. Computation Offloading Systems

In the following, we discuss computation offloading systems. This overview discusses a subset of existing approaches and is not meant to be exhaustive. Multiple systems are similar to the ones presented here and, thus, will not be discussed. We present multiple approaches and categorize them according to the dimensions identified in Section 2.2. However, the scope of the presented systems is highly heterogeneous. Some approaches present high-level visions of offloading concepts that have changed the research agenda and have given rise to new offloading paradigms. One example of this category are *Cloudlets* by Satayanarayanan *et al.* [44]. Others discuss their offloading architectures in detail and provide technical insights in their implementations such as their application partitioning algorithms and offloading decision engines. They provide evaluation results of prototypical implementations in a laboratory or real-world testbed [68, 88]. Still others describe offloading systems that are or have been actively deployed. These systems have proven their applicability in real-world, large scale settings. Published articles often contain technical details of these systems as well as usage statistics. Examples of these approaches are *BOINC* [9] and HTCondor [89]. In the following, we discuss the main foci and contributions of the different publications in chronological order. We summarize the findings in Table 3.1 at the end of this section.

*HTCondor* [24, 89] is one of the earliest and most popular systems for distributing computational workload among multiple, geographically distant computing clusters. The system is mostly beneficial for coarse-granular tasks that run for multiple hours and do not have any local dependencies. *HTCondor* gathers CPU cycles from idle workstations that agree to share their resources to a distributed supercomputer. The system is based on the *Globus* [90] toolkit that provides tools to distribute parallel computations across networked devices. *HTCondor* adds the task distribution layer on top of this toolkit and facilitates a resource-aware scheduler for tasks within a grid computing environment.

*XtremWeb* [60] is an early desktop grid and volunteer computing approach that aims to harness many unused computing resources from end user devices to build a very large parallel computer. The goal is to integrate different kinds of devices such as server farms and personal desktop computers. In *XtremWeb*, volunteers register to an administration server and install a Java-based worker software which is able to run unmodified applications. Registered application developers can spawn their own applications through a public interface which allows them to spontaneously make use of a large distributed computing infrastructure. The partitioning of the applications has to be performed manually. Fault-tolerant scheduling is performed in a FIFO scheme but can be configured dynamically to more complex strategies.

*Spectra* [74] is a partitioning and offloading system for mobile devices. Application programmers have to explicitly specify methods for application partitioning. The offloading decision, however, is performed automatically at runtime. *Spectra* takes multiple competing goals into consideration while dynamically making the offloading decision. These goals are minimizing the execution time and energy consumption on the mobile device while maximizing the application quality.

*Spectra* monitors the state of the remote servers and selects the most suitable servers based on the predicted resource demand of the current task.

*OurGrid* [91] is a peer-to-peer resource sharing system that allows reciprocal sharing of computational resources among end-user devices. Each peer in the system can act as a resource consumer and a resource provider at the same time. A decentralized scheduling protocol allocates independent tasks of parallel bag-of-tasks applications to peers with idle resources.

*Entropia* [8, 23] is an architecture for desktop grid systems that allows to incorporate large numbers of end-user devices. The system wraps unmodified Win32 executables into a sandboxed virtual machine to ensure the unobtrusiveness of applications. Thus, the state of the executing device remains unchanged after the execution of an application. End-users submit computational tasks to a job manager that splits up the workload into multiple subjobs which are then scheduled to Entropia client machines by a central subjob scheduler. The partitioning of the job requires an application-dependent pre-processor and might be implemented by splitting up the list of input parameters.

*BOINC* [9] is the most popular computation offloading and volunteer computing system. The system allows scientists to create and operate large computing projects where volunteers can share private resources to run so-called workunits of this project in their local *BOINC* clients. The *BOINC* system facilitates the distribution of the workunits and handles the collection of results. Participants are not paid for their contributions but can track their performance in global leaderboards.

*Aneka* is a Platform-as-a-Service implementation that integrates desktop grid resources with traditional cloud computing platforms to support QoS demands such as deadlines for scalable applications [61, 64, 92]. The *Aneka* container that abstracts from the underlying hardware is installed on each node and supports multiple platforms. *Aneka* implements a SOA and supports four different programming models as abstractions for distributed applications. The integration of cloud resources into desktop grids allows *Aneka* to scale up and down depending on the requirements of the application.

*Hyrax* [73] provides an infrastructure for sharing data and computational resources in a mobile device cloud. The system uses the Hadoop framework [93], an open-source implementation of MapReduce [94], ported to an Android platform. The focus of *Hyrax* are data-centric tasks that are well suited for the Hadoop environment. A central scheduler manages data and job distribution, whereas the mobile devices communicate in a peer-to-peer fashion.

In [44, 95, 96], Satyanarayanan *et al.* introduce *Cloudlets*. The idea of *Cloudlets* is to move powerful computational resources typically known from public cloud providers closer to the physical location of mobile devices. These computers or small clusters of computers are deployed in public spaces like public Wi-Fi networks that are available for users in the vicinity. *Cloudlets* can be used in a similar way by owners of mobile devices that run computationally intensive applications. The execution of the application can be offloaded by virtual machine migration to the nearby resource. After the execution, the *Cloudlet* discards the virtual machine and returns to its original state. *Cloudlets* have influenced multiple code offloading systems and represent a very early example of edge computing systems.

*MAUI* [68] is one of the most prominent examples of code offloading systems and serves as a benchmark for multiple other approaches. In their paper, Cuervo *et al.* introduce a fine-grained approach for code offloading that allows to minimize the execution time and energy consumption on mobile devices. Application programmers have to annotate the source code of their .NET applications to identify methods that qualify to be offloaded to a MAUI server which holds a copy of the application executables. An optimization framework decides whether a method should be offloaded given the current networking environment.

*Cuckoo* [66] is an offloading framework for Android applications. Application users can make use of remote resources running a Java virtual machine such as laptops, home servers, and cloud resources. Application developers need to define interfaces for the computationally intensive parts of the code. The *Cuckoo* framework generates method stubs for the remote execution that the developer can fill with either the same or an advanced version of local implementation. This allows to run more powerful algorithms on the remote devices which could increase the application's fidelity. The simple offloading decision always suggests a remote execution if the remote resource can be reached.

*CloneCloud* is a virtual machine migration system that allows mobile devices to offload parts of unmodified applications to powerful cloud resources [58]. A static analyzer and a dynamic profiler automatically partition the application offline. An optimization solver decides at runtime which parts are executed locally and which ones are executed on remote resources. The goal is to minimize the expected cost in terms of execution time and energy consumption on the mobile device. The mobile device and the cloud resource are tightly coupled. Thus, no task allocation strategies are required.

In [97], Abolfazli *et al.* introduce *MOMCC*, a market-oriented architecture for mobile cloud computing. The authors argue that WAN latencies can be avoided by simulating a nearby cloud computing platform with a group of mobile phones that host services in the vicinity of mobile devices. Service hosts get paid for the execution of services to encourage participation in the system. The approach builds upon a service-oriented architecture to bundle multiple fine grained services into a new complex functionality which can be published, discovered, and hosted by a (mobile) device.

*COCA* offers code offloading to clouds using aspect oriented programming (AOP) for Java applications running on Android mobile devices [70]. Application programmers do not have to make changes neither in the application binary nor in the original source code. Instead, *COCA* developers can manually decide which objects or functions to offload or they can leave this decision to a static or dynamic profiler. A cloud instance that holds a copy of the application can be used as offloading target. The scheduling happens in a static way to those dedicated cloud instances.

*COMET* [88] allows unmodified multi-threaded applications to offload one or multiple threads to a dedicated server. The offloaded threads run in parallel on the mobile device and the remote server on modified Dalvik Virtual Machines. As soon as the server has finished the execution of a thread, it informs the mobile device which terminates the local execution. A distributed shared memory keeps the states of both virtual machines synchronized and allows the mobile device to recover from network failures. The memory synchronization works in a lazy manner which minimizes the state transfer between the mobile device and the server. The offloading decision is based on the the current networking environment.

*ThinkAir* [69, 98] provides method-level computation offloading for mobile devices by smartphone virtualization in the cloud. *ThinkAir* provides a library that application programmers can use to annotate offloadable methods in their Android applications. These methods are compiled for the x86 platform using the Native Development Kit (NDK). Hardware, software, and network profilers monitor the context for the execution controller which performs the offloading decision optimizing execution time, energy conservation, and cost for cloud resources.

*MACS* [99] is a middleware that allows to distribute the execution of Android applications between mobile devices and cloud resources. Application developers have to follow the service-oriented structure of Android applications as only these services can be offloaded. *MACS* makes modifications of Java files in the pre-compile stage to create service proxies for the mobile device and service stubs for the cloud instance that runs a Java execution environment. A context-aware offload manager determines at runtime which services are to be offloaded and which ones are executed locally to minimize the execution time and energy consumption.

*MAPCloud* [100] implements a 2-tier cloud architecture for computationally intensive mobile applications. The architecture includes local edge resources as well as scalable remote cloud instances. A cloud resource allocation heuristic decides at runtime whether the code is sent to nearby devices or the remote cloud. Three Quality of Service parameters - price, power, and delay - can be specified. Offloading is performed at the granularity of Android services.

*Serendipity* [101] is an edge computing system that allows mobile devices to offload their computational workload to other mobile devices in their vicinity. The main focus of the system is to exchange computational jobs between mobile devices that have different movement patterns and might not be connected at all times. Jobs are defined as Directed Acylcic Graphs that consist of multiple, possibly parallel tasks. A job profiler initiates a job multiple times with different input data to create an execution profile. This execution profile is then used by the job scheduler to decide whether or not to offload a task to a nearby mobile device to minimize job execution time and energy consumption.

Angin and Bhargava [102] discuss an agent-based offloading framework for mobile cloud computing. The framework allows migrating computational workload in the form of mobile agents to cloud resources. The statical partitioning of the

the process has to be done manually by rewriting the source code in an offline phase. A central cloud directory service performs the scheduling of the application partitions to JADE [103] agent containers running on cloud instances.

*LibWater* [104] is an approach for programming distributed applications for heterogeneous computing systems including multi-core CPUs and GPUs. Applications have to follow the OpenCL programming paradigm and implement commands from the *libWater* C/C++ library-based extension. As OpenCL natively supports parallel executions, *libWater* applications can easily be split across multiple devices and be scheduled efficiently. A novel device query language facilitates resource management and discovery.

*AMCO* [105] aims at reducing the energy consumption of mobile applications through dynamic code offloading. Developers annotate their centralized Android applications to specify parts that are likely to consume a lot of energy. *AMCO* rewrites the binary of the application to allow dynamic offloading at runtime based on the current execution environment. The underlying hardware and software stacks remain unmodified. *AMCO* aims at minimizing the state transfer between the local and the remote device.

*Jade* [106] is a class-level computation offloading system that performs dynamic offloading decisions to minimize energy consumption. Developers use the *Jade* programming model library to implement classes that qualify for offloading. A profiler monitors the status of the local device, the wireless connection, and the offloadable code. To estimate resource consumption, the code is invoked multiple times with random inputs. At runtime, the *Jade* optimizer decides which code to offload to the server and whether Wi-Fi or Bluetooth is used for data transfer.

*Nebula* [62] is a distributed edge computing system for data intensive applications. Volunteers can contribute their data storage or computational power to a global computing platform. Applications are built as native client executables for the Google Chrome web browser and may consist of multiple jobs which, in turn, may contain multiple parallel tasks [107]. *Nebula* provides a web-based front-end where volunteers can register to share their resources and application developers can upload their executables. A central scheduler allocates tasks within a pool of volunteer nodes using a locality-aware scheduling algorithm to leverage the geographical distribution of data among the peers.

Cheng *et al.* propose a just-in-time code offloading system for wearable computing [108]. The system is structured as a three-layer architecture. Wearable devices can offload computation to either local mobile devices via short range communication or to remote cloud resources by using the local devices as intermediaries. A scheduling optimization algorithm based on the idea of genetic algorithms determines which part of the task gets to be executed on the wearable device, the local mobile devices, and the cloud resources.

*Kahawai* [109] is a GPU offloading system that provides high-quality gaming on mobile devices. Instead of general computational workload, the system offloads the rendering of video frames to perform collaborative rendering. In this approach, not the entire content is rendered either locally or remotely but the workload is split up between the two sides. Two different techniques are deployed. Delta encoding renders a low quality frame on the mobile device and receives information about the changes from the previous high quality frame from the server. This combination results in a high quality frame. I-frame rendering renders some high quality frames on the mobile devices whereas the remaining frames are rendered on a server.

*FemtoClouds* [110] provides an edge cloud service by leveraging the computational power of nearby mobile devices. Similar to the idea of *Cloudlets* [44], *FemtoClouds* provide computing resources that can be utilized in an opportunistic manner. One of the mobile devices act as control device which accepts tasks and schedules them to devices that have the *FemtoClouds* client service installed. The resources of these devices are shared for direct monetary compensation. Scheduling is optimized for increasing the rate of successfully executed tasks by taking into account that devices might leave the system during the task execution.

*Dust* [111] is a lightweight device-to-device offloading framework for wearable computing. Java application programmers annotate the source code to identify offloadable code blocks. Applications automatically submit these partitions to the *Dust* service which decides in a FIFO order whether or not to offload them. A profiler monitors the variables of the task and the computation environment. *Dust* supports offloading to nearby mobile devices, cloudlets, or remote clouds. A task scheduler makes the offloading decision to minimize energy consumption, CPU usage, and execution time.

Li *et al.* introduce mobile code offloading with minimal state transfer to remote cloud instances [112]. The approach allows unmodified Android applications to offload computationally intensive methods to cloud resources without any programmer interaction. Instead, the system performs a byte code analysis to create an execution model of the application. An offline parser can then determine the required state transfer for each execution path which is minimized dynamically at runtime. The offloading decision is based on the execution model and historic records about the previous executions.

*CloudAware* [113–115] introduces a self-adaptive, context-aware system for mobile edge and cloud computing. *CloudAware* provides an API that allows developers to implement distributed applications with only small changes in the Android source code. Context-awareness is facilitated through a context manager that learns from historic behavior patterns of mobile devices and predicts the context of future environments such as network connectivity and the execution time of offloaded tasks.

*Avatar* [116] is a code offloading system that allows mobile devices to leverage cloud resources. End-user owned virtual machines in the cloud, the so-called avatars, run the same operating system as the mobile device. Offloading is performed by virtual machine migration. Application developers use the *Avatar* API to structure their applications and facilitate distributed execution on the mobile devices and virtual machines in the cloud.

*COARA* [117] implements an aspect-oriented code offloading approach for Android applications with AspectJ. Application programmers need to annotate methods and classes in the source code to identify the parts that qualify for being offloaded. The Android operating system remains unmodified. *COARA* aims at reducing the state transfer between the client device and the remote server to reduce the execution time and bandwidth consumption. In order to achieve this, the server uses object proxies which are only loaded on demand. Scheduling is of no importance as client and server are tightly coupled.

The *Mobile Cloud Offloading Architecture* (MOCA) [119] utilizes software defined networking (SDN) and in-network cloud platforms to realize low-latency offloading in mobile networks. In-network cloud platforms are dedicated cloud infrastructures that are maintained by the mobile network provider. The approach follows the

| Project | Year | Part. | | | Dec. | | Granularity | | | | | Host | | | Target | | | | | Goal | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Explicit | Annotated | Automated | Static | Dynamic | Application | Thread/Service | Class | Job/Task | Method | Static | Mobile | Wearable | Mobile Device | Static Device | Dedicated Server | Cloudlet | Cloud/Grid | Energy | Execution Time | Cost | Fidelity | Scalability |
| HTCondor [24, 89] | 1988 | ● | | | ● | | | | | ● | | ● | | | | | | | ● | | | | | ● |
| XtremWeb [60] | 2001 | ● | | | ● | | ● | | | | | ● | | | | ● | | | ● | | | | | ● |
| Spectra [74] | 2002 | | ● | | | ● | | | | | ● | | ● | | | ● | | | | ● | ● | | ● | |
| OurGrid [91] | 2003 | | | | | | | | | ● | | ● | | | | ● | | | | | | | | ● |
| Entropia [8, 23] | 2003 | ● | | | ● | | ● | | | | | ● | | | | ● | | | | | | | | ● |
| BOINC [9] | 2004 | ● | | | | | | | | ● | | ● | | | ● | ● | | | | | | | | ● |
| Aneka [61, 64, 92] | 2009 | ● | | | | ● | | | | ● | | | ● | | | ● | | | ● | | | ● | | ● |
| Hyrax [73] | 2009 | ● | | | | ● | | | | ● | | | ● | | ● | | | | | ● | ● | ● | | |
| Cloudlets [44] | 2009 | ● | | | | ● | ● | | | | | | ● | | | | | ● | | | ● | | | ● |
| MAUI [68] | 2010 | | ● | | | ● | | | | | ● | | ● | | | | ● | | | ● | ● | | | |
| Cuckoo [66] | 2010 | ● | | | ● | | | | | | ● | | ● | | | ● | | | ● | | | | ● | |
| CloneCloud [58] | 2011 | | | ● | ● | | | ● | | | | | ● | | | | | | ● | ● | ● | | | |
| MOMCC [97] | 2012 | | | | | ● | | ● | | | | | ● | | ● | | | | | | | | | |
| COCA [70] | 2012 | | ● | | ● | | | | | | ● | | ● | | | | | | ● | ● | ● | | | |
| COMET [88] | 2012 | | | ● | | ● | | ● | | | | | ● | | | | | | | ● | ● | | | |
| ThinkAir [69, 98] | 2012 | | ● | | | ● | | | | | ● | | ● | | | ● | | | | ● | ● | ● | | |
| MACS [99] | 2012 | | | ● | | ● | | | | | ● | | ● | | | | | | ● | ● | ● | | | |
| MAPCloud [100] | 2012 | ● | | | | ● | | ● | | | | | ● | | | ● | | | ● | ● | ● | ● | | |
| Serendipity [101] | 2012 | ● | | | | ● | | | | ● | | | ● | | ● | | | | | ● | ● | | | |
| Angin *et al.* [102] | 2013 | ● | | | | ● | | | ● | | ● | | ● | | | | | | ● | ● | ● | | | |
| LibWater [104] | 2013 | ● | | | | ● | | | | | ● | ● | | | | ● | | | ● | | | | | ● |
| AMCO [105] | 2013 | | ● | | | ● | | | ● | | | | ● | | | | | | ● | ● | | | | |
| Jade [106] | 2014 | ● | | | | ● | | | ● | | | | ● | | | | ● | | | ● | | | | |
| Nebula [62] | 2014 | ● | | | | ● | | | | ● | | ● | | | | ● | | | ● | | ● | | | ● |
| Cheng *et al.* [108] | 2015 | ● | | | | ● | | | | ● | | | | ● | ● | | | | ● | | ● | | | |
| Kahawai [109] | 2015 | ● | | | ● | | ● | | | | | | ● | | | | ● | | | | | | ● | |
| FemtoClouds [110] | 2015 | | | | | ● | | | | ● | | | ● | | ● | | | | | | | | | ● |
| Dust [111] | 2015 | | ● | | | ● | | | | | ● | | | ● | ● | | | ● | ● | ● | ● | | | ● |
| Li *et al.* [112] | 2015 | | | ● | | ● | | | | | ● | | ● | | | | | | ● | ● | ● | | | |
| CloudAware [113–115] | 2015 | ● | | | | ● | | | ● | | | | ● | | ● | | | | ● | ● | ● | | | |
| Avatar [116] | 2015 | ● | | | | ● | | | | ● | | | ● | | | | | | ● | ● | ● | | | ● |
| COARA [117] | 2016 | | ● | | ● | ● | | | ● | | ● | | ● | | | | ● | | | | ● | | | |
| MpOS [118] | 2017 | | ● | | ● | | | | | | ● | | ● | | | | | ● | ● | | ● | | ● | |

Table 3.1.: Existing offloading approaches. The majority of systems focuses on energy savings when offloading from mobile devices to either dedicated servers or cloud and grid resources. No system considers multiple host device types. Several early approaches offload from static devices to other static devices to use remote resources and allow the execution of computationally complex applications.
(Part.: Partitioning, Dec.: Offloading Decision)

idea to extend the existing mobile infrastructure instead of forwarding offloaded traffic through the Internet backbone to regular cloud instances.

*MpOS* [118] is a method-level offloading framework for multiple mobile platforms. The system allows the integration of remote public cloud resources as well as resources in the local network such as *Cloudlets* [44]. The partitioning is based on manual annotations in Java or C# source code. At runtime, *MpOS* intercepts each annotated method and checks whether the method qualifies for offloading and whether offloading is beneficial. The dynamic decision system can be configured by the developer and accepts rules such as a maximum round trip time or a particular connection type. There is no automated offloading decision making.

## 3.2. Context-Aware Task Schedulers

To increase the performance of the task allocation decisions, context-aware schedulers need to take the status of the environment into account. In the literature, we have identified multiple context dimensions that we classify into the categories offloading target, task, network, and miscellaneous. We discuss the context dimensions on the examples of selected scheduling approaches from the literature. We further discuss the reliability of the approaches in terms of failure handling. A comprehensive summary about context-aware schedulers can be found in Table 3.2 at the end of this section.

### Offloading Target

The resource providers in the system are the offloading targets for task schedulers. The success and performance of the offloaded task highly depend on the context of these resource providers. As offloading systems can include all types of devices, the heterogeneity among these devices can become large. Thus, a careful selection of suitable providers is vital for the performance of the distributed application.

The computational power of the offloading target determines the remote execution time of the task, because the resource providers can vary significantly in the number of CPUs and their processing power. By taking the processing power into account, schedulers can estimate whether offloading is beneficial compared to a local execution. They can also attempt to meet a task's deadline or reduce the execution time of a task. *NetSolve* [120], for example, uses a benchmark program to obtain the Kflop/s rate on an idle machine. The result is then used to determine the most suitable provider. Similar concepts can be found in early approaches such as *AppLeS* [121] as well as more recent schedulers, like *mCloud* [122]. Schedulers can also take the hardware configuration into account. Resource providers might be equipped with specialized hardware such as GPUs and multi-core processors that can be useful to execute highly parallel tasks. Tasks that can benefit from parallelism can be offloaded to providers that are well suited for parallel executions.

When providers get overloaded, they might queue tasks, slow down or might even reject resource requests. To avoid delays, most schedulers take the current load of a device into account. Several schedulers actively perform load balancing to avoid the concentration of workload on single devices. Load balancing can be performed by work stealing as in *Sparrow* [123] or by resource monitoring as in *Utopia* [124].

Resource providers can be either centrally administered cloud resources or end-user owned edge devices. In contrast to static grid and cloud resources, the *reliability* of edge devices varies. These end-user administered devices might enter and leave the system at any time and, thus, can only be used in a best-effort manner.

Providers which leave the system during the execution of a task or drop the task for arbitrary reasons can harm the performance of the distributed application. Depending on the importance or urgency of a task execution, context-aware schedulers can select the right level of reliability among the available providers. The *Aneka* scheduler, for example, uses stable resources to ensure that task deadlines are met [92].

Owners of computational devices might ask for a monetary compensation for sharing their resources. Costs have fixed rates or vary depending on demand and supply, such as in [125]. Resources might also be offered for free to either everybody or to selected consumers. As a general assumption, a good scheduling decision keeps the cost of resource usage low. Examples of cost-aware task schedulers can be found in *Nimrod/G* [126], *GrADS* [127], *GridWay* [128], *SpeQuloS* [129], *ThinkAir* [69], and *COSMOS* [130].

### Task

Schedulers need to consider the task characteristics to decide whether and whereto the task should be offloaded. As the offloading process itself requires time and energy, the size is an important factor. Tasks grow with the amount of data that is associated with the execution of the task, because this data has to be forwarded to the remote host as well.

The majority of schedulers also considers the complexity of the task as an input for the offloading decision. As the complexity of the task typically is unknown before the execution of the task, schedulers need to estimate the runtime. *PUNCH* [131], for example, implements a learning approach based on polynomial regression to predict the resource usage for every single run. *Jade* [106] uses a program profiler that executes a program multiple times with random inputs and uses the average execution time as an estimator for a task's complexity. *HTCondor* [24] uses so-called *ClassAds* to define the characteristics and requirements of a task manually. Schedulers in *MAUI* [68] and *CloneCloud* [58] use a combination of a static profiler that analyzes the code offline and a dynamic profiler that provides a cost model for different input sets.

Though less discussed, the memory requirements of applications are also taken into account by many context-aware schedulers. Among those are *PBS* [132], *SLURM* [133], and *Haizea* [134]. The scheduler in *OLIE* [85] performs an extensive byte code analysis and estimates the memory requirements by measuring the object sizes and the frequency in which they are called.

As energy conservation on mobile devices is a common goal of multiple computation offloading systems, some schedulers directly estimate the energy consumption of the task execution. In *Serendipity* [101], *MAUI* [68], and *CloneCloud* [58], an offline job profiler creates a complete profile about the job's resource requirements including its energy consumptions. This profile is then used to make the offloading decision.

Systems that provide hard or soft real-time guarantees need to take the deadline requirements of tasks into account. *Mars* [135] is a fault tolerant distributed real-time system that guarantees hard deadlines even under peak load. However, the system assumes an underlying deterministic medium access. *Aneka* [61] also takes deadlines into account. When sufficient time is left, unreliable resources can be selected for execution. As soon as the risk of missing a deadline increases, stable resources are selected. Further systems that allow for real-time execution are, for example, *Nimrod/G* [126], *LODCO* [136], and *EPCO* [137].

### Network

Whereas the target device profile and task characteristics mainly determine the execution time of a task, the network connection directly affects the duration of the offloading process. A low bandwidth and a long delay both increase the overall execution time and make task offloading less attractive. As these parameters fluctuate, context-aware schedulers need to constantly monitor the network connection. About half of the selected schedulers consider bandwidth as well as delay for the offloading decision. However, hardly any information can be found how these parameters are monitored. *OLIE* [85] and *BreadCrumbs* [138] use periodic pings to measure delays. To estimate the bandwidth, *BreadCrumbs* downloads a file as fast as possible from a well-known server. The estimates are used to forecast future connectivity.

Only few schedulers take the stability and availability of the network connection into account. One noteworthy example is *Serendipity* [101] which predicts the encounter of mobile devices in the future. Only when these devices are in proximity they can exchange tasks and results.

### Miscellaneous

Some schedulers in task offloading system take context information into account that go beyond the categories of target device, task, and network conditions. *MapReduce* [94], for example, takes the location of the intermediate result data into consideration when scheduling *reduce* tasks. *Entropia* [8] uses file caching to eliminate redundant data transmissions. Tasks can then be scheduled to devices where the data is already present. *ENDA* [139] predicts user mobility based on historic movement traces to determine the optimal offloading strategy.

Multiple schedulers also base the offloading decision on the current battery level of the local device. *Spectra* [74], for example, computes the remaining battery lifetime and adapts the importance of energy conservation accordingly. *LODCO* [136] presents a scheduler for edge devices using energy harvesting that computes, based on the current battery level, whether a task can be executed before the battery of the device runs out.

| Project | Year | Target | | | | | Task | | | | | Netw. | | | Misc. | | Reliability | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Reliability | Cost | Performance | Load Balancing | Load | Size | Complexity | Memory | Energy | Deadline | Bandwidth | Delay | Stability | Data Location | Host Energy | Fault-Awareness | Fault-Tolerance | Fault-Avoidance | Redundancy | Checkpointing |
| HTCondor [24, 89] | 1988 | | | ● | ● | ● | ● | ● | ● | | | | | | | | ● | ● | | | ● |
| Mars [135] | 1989 | | | | | | ● | ● | | | ● | | | | | | ● | ● | | ● | |
| Sprite [140] | 1991 | | | | | ● | ● | ● | | | | ● | ● | | | | ● | ● | ● | | |
| Utopia [124] | 1993 | | | ● | ● | ● | ● | ● | ● | | | | | | | | ● | ● | | | |
| PBS [132] | 1995 | | | ● | | ● | | | ● | | | | | | | | | | | | |
| NetSolve [120] | 1996 | | | ● | ● | ● | ● | ● | | | | ● | ● | | | | ● | ● | ● | | |
| AppLeS [121] | 1997 | | | ● | | ● | | ● | ● | | | ● | ● | | | | ● | ● | | | |
| Globus [90] | 1997 | | | ● | | ● | | | ● | | | ● | ● | ● | | | | | | | |
| Javelin [141, 142] | 1997 | | | | ● | ● | | | | | | | | | | | ● | ● | | | |
| Ninf [143] | 1999 | | | | | ● | | | | | | ● | | | | | ● | | | | |
| Legion [144] | 1999 | | | | | ● | | | ● | | | | | | | | ● | | | | |
| PUNCH [131] | 1999 | | | | | | ● | ● | ● | | | | | | | | | | | | |
| Nimrod/G [126] | 2000 | | ● | ● | | ● | ● | | | | ● | ● | ● | ● | | | ● | ● | | | |
| REXEC [145] | 2000 | | | | | ● | | | | | | | | | | | ● | | | | |
| Darwin [146] | 2001 | | | ● | | ● | | ● | ● | | | ● | ● | | | | ● | | | | |
| GrADS [127] | 2001 | ● | ● | ● | | ● | | ● | | | | | | | | | ● | ● | ● | | |
| XtremWeb [60] | 2001 | | | | | ● | | | | | | | | | | | ● | ● | | | |
| 2K [147] | 2002 | | | | | ● | | ● | | | ● | ● | | | | | ● | ● | | | |
| Spectra [74] | 2002 | | | ● | | ● | | ● | | ● | | ● | ● | | | ● | | | | | |
| SLURM [133] | 2003 | | | ● | | ● | | ● | ● | | | | | | | | ● | ● | | | |
| Entropia [8, 23] | 2003 | | | ● | | ● | | | ● | | | ● | ● | ● | ● | | ● | ● | | ● | |
| OurGrid [91] | 2003 | | | ● | | | | ● | ● | | | | | | | | ● | ● | | | |
| OLIE [85] | 2003 | | | | | ● | ● | | ● | | | ● | ● | | | | | | | | |
| GridWay [128] | 2005 | | ● | | | ● | | ● | ● | | | | | | | | ● | ● | | | |
| MapReduce [94] | 2008 | | | | | ● | ● | | | | | | | | ● | | ● | ● | | ● | |
| Haizea [134] | 2008 | | | | | ● | ● | ● | ● | | | | | | | | ● | ● | ● | | ● |
| BreadCrumbs [138] | 2008 | | | | | | | | | | | ● | ● | | | | | | | | |
| Aneka [61, 64, 92] | 2009 | ● | | | | ● | | ● | | | ● | | | | | | ● | ● | | ● | |
| Cloudlets [44] | 2009 | | | | | ● | | | | | | ● | ● | | | | | | | | |
| MAUI [68] | 2010 | | | | | | | ● | ● | ● | | ● | ● | | | ● | ● | ● | | | |
| CloneCloud [58] | 2011 | | | ● | | ● | | ● | | ● | | ● | ● | | | | | | | | |
| SpeQuloS [129] | 2012 | ● | ● | | | | | | | | | | | | | | | | | | |
| ThinkAir [69, 98] | 2012 | | ● | ● | | | | ● | | ● | ● | ● | ● | | | ● | ● | | | | |
| Serendipity [101] | 2012 | | | | | ● | | ● | | ● | | ● | ● | ● | | ● | | | | | |
| ENDA [139] | 2013 | | | | ● | ● | ● | | | | | ● | ● | | ● | | ● | ● | | | |
| Sparrow [123] | 2013 | | | ● | ● | ● | | | | | | | | | | | ● | ● | | | |
| C2OF [148, 149] | 2014 | | | | | | ● | ● | | | ● | ● | ● | | | ● | ● | ● | | | ● |
| COSMOS [130] | 2014 | | ● | | | | | ● | | | | ● | ● | | | | | | | | |
| Jade [106] | 2014 | | | | | | ● | ● | ● | | | ● | ● | | | ● | | | | | |
| LODCO [136] | 2016 | | | | | | ● | | | | ● | ● | ● | | | ● | | | | | |
| Guo *et al.* [150] | 2016 | | | | | | ● | | | | ● | ● | ● | | | ● | | | | | |
| EPCO [137] | 2016 | | | ● | | | ● | ● | | | ● | ● | ● | | | | | | | | |
| mCloud [122] | 2017 | | | ● | | | ● | ● | ● | | | ● | ● | | | ● | ● | ● | | | |

Table 3.2.: Context-aware task scheduling approaches. The majority of approaches takes at least the context of either the target device, the task, or the network into account when making offloading decisions. Beyond that, only few context dimensions are considered. Whereas most schedulers are fault-aware, only few provide mechanisms for fault-avoidance or compensate for aborted task executions. (Netw.: Network, Misc.: Miscellaneous)

### Reliability

The distributed nature of computation offloading systems make these systems vulnerable to several types of failures. Resource providers can leave the system, abort the computation, or might lose network connection. Execution requests as well as results can get lost and parts of the system might be overloaded or faulty. Here, we discuss how context-aware schedulers handle failures and what strategies they use to tolerate or avoid failures.

The majority of selected schedulers provides at least fault-awareness. *Darwin* [146] is one example that provides callback mechanisms to notify applications about a success or failure of a computation. The handling of the failure, however, is up to the application developer. Other schedulers, such as in *XtremWeb* [60], provide fault tolerance by re-initiating execution requests when failures occur. Further examples are *Javelin* [141] and *2K* [147]. Fault-avoidance is, for example, implemented by *NetSolve* [120]. The scheduler receives a list of suitable execution targets that is ordered by the probability that the target device accepts an execution request. Thus, the number of unsuccessful requests is reduced. *GrADS* [127] uses resource reservation to ensure that the resources are available at the time they are required. This concept is extended in *Haizea* [134] where the authors suggest to over-reserve resources to proactively account for failures. Schedulers in *Entropia* [8], *MapReduce* [94], and *Aneka* [61] use redundancy to reduce the effect of failures. A different strategy is used by *HTCondor* [24], *Haizea* [134], and *C2OF* [148,148] that use checkpointing to compensate for aborted task executions.

## 3.3. Discussion

The analysis of existing computational resource sharing systems and context-aware schedulers has shown that neither of the existing approaches implements our vision of a holistic resource sharing system that allows the execution of tasks from arbitrary applications in a heterogeneous computation environment. Instead, the approaches are tailored to specific use cases such as volunteer computing for scientific projects [9], optimized video frame rendering [109], energy saving for mobile devices [68], integration of wearable devices [111], automated task partitioning [112], or opportunistic resource usage [44].

Each context-aware scheduling approach only considers a subset of the context-dimensions identified in Section 3.2. The reliability of the offloading target, for example, is only monitored by three schedulers which, in turn, do not take the context of the network into account.

# 4. Requirement Analysis

In this section, we identify the requirements of an holistic computation offloading system. Therefore, following Maciaszek (2007) we identify the stakeholders in such a system and discuss what each group expects from it [151]. Based on the insights from this analysis, we outline the functional and non-functional requirements of an offloading system. These requirements serve as guidelines during the design and implementation process of our proposed solution.

## 4.1. Scenario

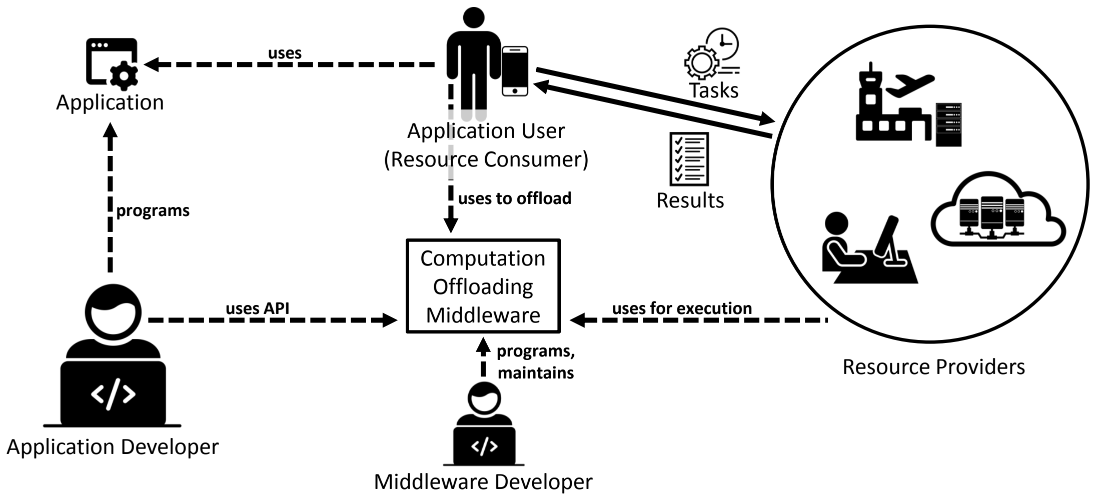


Figure 4.1.: Players in an offloading system

We use a scenario to identify the stakeholders of a computation offloading system and to discuss the interaction between the stakeholders and the system. Figure 4.1 shows all relevant players with the offloading system in the center. Before an offloading system exists, developers perform a requirement analysis, design the system, and eventually implement it. They are also responsible for the deployment of the system as well as its maintenance.

Application developers of computationally intensive applications use the offloading system to send workload to remote devices. They use the API of the middleware. Hereby, application programmers do not have to implement any of the communication that is necessary to offload the computation. They can focus on the application development and benefit from the possibility to use more resources than single devices provide. As a result, applications can become more complex and perform more sophisticated computations and at the same time might consume less energy.

These applications are installed by end users which are the resource consumers in this system as they make use of remote computing devices. For the users the offloading process remains mostly transparent and they would not notice the difference between a local or a remote execution of tasks. The application might offload tasks via the middleware to remote resources. When the remote execution is done these resources return the result to the application.

Resource providers share their computational power with resource consumers. They install the offloading middleware to provide the execution environment for tasks. Resource providers can be either device owners that contribute their idle computing cycles as in [9], computational grids or cloud structures as in [24], or dedicated servers in close proximity in public places such as airports or coffee shops as in [44]. It is the responsibility of the offloading system to match offloaded tasks with remote resources. As this matchmaking heavily depends on the current situation of the application user, the remote resources, as well as the tasks itself, the offloading system needs to take the context of these entities into account.

## 4.2. Stakeholders

In the scenario above, we have identified the key stakeholders of computation offloading systems. In related work, most approaches focus on the role of the developer [68,69] and design the system in a way that allows an easy development of distributed application. Here, we also discuss middleware developers, application users, and resource providers as stakeholders of offloading systems.

**Middleware Developers:** The developers of a computation offloading middleware are responsible to design and implement the system. They provide clear abstractions for application developers and also make it easy and intuitive for resource consumers and resource providers to use the system. The developers also need to ensure that the system can be deployed and maintained in the future. This requires a well structured documentation, testability, and adaptability. Further, over time the system might require some adjustment and extensions. Thus, it should be possible for developers to extend the system with additional features without changing the overall structure of the middleware.

**Application Users:** Users of computationally intensive applications expect their programs to run smoothly regardless of the physical capabilities of their devices. As computational offloading can help to decrease response times, reduce energy consumption, and increase the amount of available computing resources, application users are the interest group that benefits most from a computation offloading system. At the same time, the fact that the application offloads workload to remote resources should be absolutely transparent to the application users and should not require any involvement, installation, or administration. Despite the presence of communication failures, network fluctuation, or malicious resource providers they expect the application to work smoothly. Users should not be involved in the offloading process by actively taking part in the offloading decision.

In addition, the offloading system must not impair the usability of the device. Measuring the context, making offloading decisions, and sending tasks to remote devices requires computational power as well as energy. The offloading system, whose major goals are to reduce response times and battery consumption, must neither drain the device's battery nor reduce the available performance for the user of the device.

User involvement can become necessary when the relationship between application users and the providers of computational resources is relevant. For example, application users might restrict the offloading to anonymous resource providers due to security or privacy concerns as they do not want their computation or their data to be hosted on unknown devices. Further, using remote resources might come at a cost. Users should be able to define how much money can be spent and also define their preferences about timely executions and involved costs. These settings, privacy and monetary, should be accessible to the user in an understandable and non-technical way to hide the complexity of the offloading system.

Offloading computation includes the transmission of personal data to remote devices. As the data has to be transmitted and the execution takes place on potentially malicious nodes there is a risk of exposing personal or confidential information. Ideally, application users can trust the distributed execution in the same way they can trust an execution on their local device as the offloading system protects their privacy and eliminates security threads. If this is not the case, users should be at least aware of the fact that their data is transferred to remote machines. While this reduces the distribution transparency, it allows for a manual control over one's data.

**Application Developers:** Whereas application users benefit most from offloading systems, application developers are the ones that actually interact with the systems. They use the provided API to implement computation offloading into their applications. The incentive for developers to use offloading systems is to augment the amount of available resources and to make their applications more responsive and less battery hungry. Offloading requires communication between geographically distributed devices which are connected via different networking technologies. The offloading process includes the identification of the offloadable parts of the application, deciding which of these parts get offloaded, finding a remote resource for execution, marshalling and sending the program and the data, monitoring the remote execution, and finally receiving and demarshalling the execution result. All these activities need to be implemented. An offloading system should reduce the burden of implementing the offloading procedure for developers and provide clear and easy-to-use interfaces. The offloading process itself should be transparent for developers and not need additional programming effort.

In a local setting without any offloading, developers expect a task to be executed eventually. In a remote environment, multiple errors can occur during the transmission and the remote execution of the task. Thus, there is no guarantee anymore that the task will be executed at all. This is acceptable as long as the offloading system shows a consistent and predictable behavior in case of errors. Developers can then take these cases into consideration and take countermeasures such as starting a local execution as backup. The offloading system might already provide fault tolerance and handle connection losses or aborted executions autonomously.

Application developers typically know best which parts of the application require certain guarantees in terms of speed, reliability, or other non-functional requirements. Whereas some parts are rather demanding in terms of requirements others can be executed in a best-effort manner. An offloading system should allow developers to make a distinction between these parts and provide sufficient guarantees if necessary but avoid additional overhead where guarantees are not required.

**Resource Providers:** Another user group of the offloading system are device owners who share their computational power. Provided resources can be end user devices, dedicated servers in proximity, as well as centrally administered cloud resources. Executing programs from unknown sources involves a risk. Thus, for resource providers it is the primary requirement that the offloading system provides security and does not allow malicious code to corrupt their devices.

There is a vast amount of diverse devices that might serve as computational resources. An offloading system needs to be able to handle this heterogeneity in terms of hardware and software capabilities as well as other factors such as varying connectivity and availability. Resource providers expect the system to be easy to install and to maintain. Any additional effort might reduce the motivation and willingness to participate and to share resources. Further, resource providers should not be distracted by the execution of tasks. Instead, local processes should have priority and should not be slowed down by task executions for remote devices.

In systems where end users share their computational resources, incentives can be used to motivate sharing. These incentives can either be based on social interactions such as sharing resources with friends. In this case, the offloading system needs to provide the possibility to identify friendship relations among users. Other incentives such as reciprocal sharing, monetary, or quasi-monetary incentives require a reliable and accurate accounting mechanism that tracks the amount of provided and used computational power.

## 4.3. Functional Requirements

Based on the identification of the stakeholders, we extract the functional ($R_F$) and nonfunctional requirements ($R_{NF}$) of a computation offloading middleware.

Functional requirements describe what a system should be able to perform whereas nonfunctional requirements define how the system should behave [152].

**$R_F1$ - Task Offloading:** The main purpose of the system is to offload workload to remote devices. Thus, the systems needs to allow to integrate task offloading into computationally intensive user applications. Therefore, a well-structured and well-documented programming interface is required. The offloading process needs to be straightforward to use for application developers and transparent to application users. The process includes the profiling and partitioning of the application, marshalling and sending the task, as well as retrieving the results.

**$R_F2$ - Quality of Service Support:** To make the offloading system usable for as many applications as possible, the requirements of these applications need to be covered. Whereas some applications require rather strict guarantees such as reliability or real-time execution, others might work in a best-effort manner. Quality of Service (QoS) Support allows to tailor the behavior of the offloading system to the requirements of the application without adding unnecessary overhead to lightweight applications.

**$R_F3$ - Context-Aware Scheduling:** Whereas $R_F1$ focuses on the offloading process itself, context-aware scheduling is required to perform an efficient matchmaking between offloaded tasks on one side and computational resources on the other side. This offloading decision depends on multiple factors as discussed in Section 2.2.2. As the offloading decisions can optimize for various goals such as reducing response times or battery consumption, the system needs to provide an option to define these goals. It needs to acquire the context and make the decisions whether and where to offload the workload at runtime.

**$R_F4$ - Heterogeneity Support:** In distributed systems, there are multiple dimensions of heterogeneity. Application developers can write applications for multiple platforms, including powerful desktop computers and resource-poor mobile devices. Similarly, resource providers can provide resource-rich or resource-poor devices for execution. In addition, computational resources do not only include central processing units but also graphics processing units (GPUs) and field-programmable gate arrays (FPGAs). Further dimensions of heterogeneity include programming languages, network connectivity, availability, and reliability.

The offloading system needs to run on all those platforms and internally handle these heterogeneities to provide a uniform abstraction to application programmers.

**$R_F$5 - Accounting:** Resource providers might either offer their resources for free or demand any kind of compensation in return. It could also be possible that resource sharing works in a reciprocal way where resource providers turn into resource consumers and vice versa. Participants earn credits by sharing their resources which they then can spend to use remote resources themselves. For these purposes, the offloading system needs to provide means of accounting to track the amount of provided and consumed resources for each participant.

**$R_F$6 - Incentives:** In literature, there are multiple offloading approaches available that prove the technical feasibility of these systems [68, 69]. However, the success of these systems - so far - is limited to grid or volunteer computing approaches such as HTCondor [24] or BOINC [9]. One possible reason for this fact is the lack of motivation for application programmers and resource providers to participate in offloading systems. Therefore, offloading systems need to motivate resource owners to share their computational power. These incentives can be monetary or non-monetary in nature.

## 4.4. Nonfunctional Requirements

Besides the functional requirements discussed above, the system also needs to fulfill nonfunctional requirements. Sometimes also called behavioral requirements [153] they do not specify implementation but define properties and constraints that effect the development or the deployment of a system or both.

**$R_{NF}$1 - Extensibility:** Even though an offloading system has been designed and implemented thoroughly, there might be the need for modifications or extensions in the future. This might be the case because of technological advances, changed user demands, or technical improvements. Thus, the system needs to be extensible to developers. Tanenbaum and Van Steen (2007) define extensibility of a system as the ability to replace parts of the system without breaking any functionality to seamlessly add new features to the system [154]. An extensible approach allows developers to maintain and extend the offloading system.

**R$_{\mathbf{NF}}$2 - Scalability:** The number of participants in distributed systems is typically not limited. An illustrative example is the world wide web with its billions of users worldwide. Thus, one important requirement of distributed systems in general - and the offloading system in particular - is scalability. Neuman (1994) identifies three types of scalability: numerical, geographical, and administrative [155]. Numerical scalability means that the performance of the system remains stable with a growing number of participants. Geographical scalability allows participants to be far apart from each other while using the system. Administrative scalability is the ability of a system to be maintainable even though multiple independent administrative entities need to interact. The offloading system needs to be scalable in all three dimensions as a large number of geographically distributed users might participate and the system might be administered by multiple organizations.

**R$_{\mathbf{NF}}$3 - Performance:** The performance of a distributed system is the key indicator for its quality. Even though a system fulfills all functional requirements, it will most likely not become successful if it does not perform the tasks efficiently. Performance can be measured in multiple ways. In the offloading system, possible metrics are the overhead that is introduced by the offloading process, the decrease in response times, the energy consumption, as well as the additional network traffic. Ideally, the offloading system decreases the execution times of applications while reducing the energy consumption on the local device.

**R$_{\mathbf{NF}}$4 - Robustness:** In distributed systems, multiple unexpected situations can occur. Devices might fail, get disconnected, or perform a malicious behavior. The systems need to remain robust, that is, it continues to work despite errors in the system [156]. In the offloading system, errors can happen during the transmission and execution of task and results. Robustness does not mean that the execution of tasks is guaranteed. In fact, it is only required that the system shows a predictable and consistent behavior in case of failures and does not crash because of unexpected events.

**R$_{\mathbf{NF}}$5 - Security and Privacy:** As multiple devices work together in the offloading system, these devices must be protected from attacks. The tasks that are sent for remote execution must not be able to perform malicious activities on the provided resource. Further, the offloading system should not open any ports for attacks that could be used to install malware on the device or control the device remotely. As application code as well as user data is transferred between devices the system might be vulnerable to attacks and confidential or private data might be revealed. The offloading system needs to protect the users' privacy.

# 5. The Tasklet System

The overview of systems for computation offloading in Chapter 3 has shown that an extensive amount of research has been conducted on this problem. Multiple offloading systems exist that all ease the distribution of computationally intensive applications. While each system focuses on one or a few aspects of the overall topic, there is, to the best of our knowledge, no system that meets all requirements discussed above. Thus, in this chapter, we develop a system for computation offloading that fulfills both, the functional as well as the nonfunctional requirements.

We start by outlining the overall concept of our system. In this high level overview, we introduce the main ideas of our system and define our view on computation offloading. We then present the design of our system. We introduce each component of the system and illustrate how these components work and interact with each other. Parts of this chapter are based on [157][1].

## 5.1. Design

In this section, we introduce our approach of a distributed computation offloading environment - the Tasklet system. We will explain later, what a *Tasklet* is. We have discussed the overall idea of the Tasklet system, i.e., our vision, in Section 1.1. According to this understanding, computation becomes a commodity and can seamlessly be exchanged between computational devices. Whenever there is something to compute, each computational device is able to perform this computation. Resource limitations of local devices vanish as for computationally intensive applications remote resources can be used just as well. To accomplish this goal, it is necessary to have a closer look on both parts of the problem. On the one hand, there are computationally intensive applications that require more

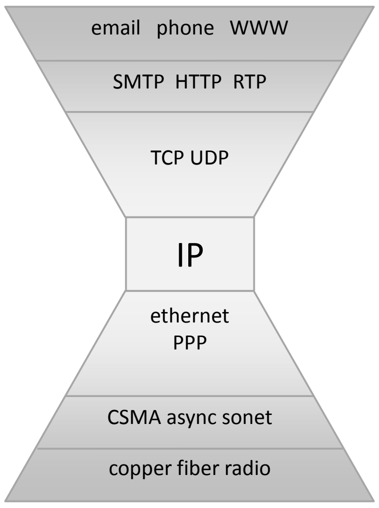
[1] [157] is joint work with D. Schäfer, S. VanSyckel, J. M. Paluska, and C. Becker

resources than a local device typically provides. On the other hand, there are heterogeneous resources that might provide computational power. Bringing these two sides together is the goal of the Tasklet system.

Computationally intensive applications can be found in many domains, including speech recognition, natural language processing, computer vision and graphics, machine learning, augmented reality, as well as planning and decision-making [44]. Applications in these domains can benefit from an augmented pool of available resources. However, applications are written in different programming languages which reduces the ability to leverage remote resources. A program written in C/C++, which is compiled for a Windows platform, can hardly be executed on a smartphone running Android and vice versa. Even for more platform independent languages such as Java, the degree of portability is limited. In addition, different applications have different requirements regarding execution guarantees. Whereas some applications require a reliable real-time execution, others do not require any guarantees. The Tasklet system should integrate all these different applications and provide a common abstraction for computation that allows these heterogeneous applications to offload workload to remote resources.

The computational resources, however, are also everything but homogeneous. First, they differ widely in their hardware configuration, including their central processing units, graphics processing units, and memory. Powerful, high-performance desktop computers are as present as resource-poor mobile devices. Second, they run different operating systems and have various software capabilities. This makes it hard to make assumptions about the provided functionalities. Third, devices vary in their network connectivity and availability. Stable and centrally administered cloud resources are typically connected via fast fiber-optic cables and are available most of the time. In contrast, the connection of end-user devices is often slow and unstable and they might leave a system at any time. Despite all these obstacles, all these resources should be able to contribute their computational power to the Tasklet system.

As a solution, we design the Tasklet system as a layered architecture with a well-defined abstraction for computation that we call Tasklet. A Tasklet is a self-contained unit of computation which means that it contains everything (including code and data) that is required to schedule and execute it as well as to return the results of the execution to the application. Applications can create Tasklets



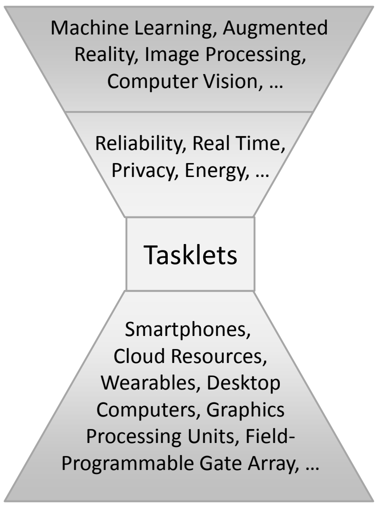

Machine Learning, Augmented Reality, Image Processing, Computer Vision, ...
Reliability, Real Time, Privacy, Energy, ...
Tasklets
Smartphones, Cloud Resources, Wearables, Desktop Computers, Graphics Processing Units, Field-Programmable Gate Array, ...

Figure 5.1.: Left: Hourglass model of the Internet, Right: Hourglass model of the Tasklet system

and pass them to the Tasklet middleware that takes care of the distribution. Resource providers run an execution environment for Tasklets that allows them to execute these Tasklets. This solution draws from the TCP/IP model of the Internet, in which multiple, heterogeneous applications run on a variety of devices. In the Internet, the Internet protocol (IP) serves as the abstraction that allows to integrate all these applications and devices. Each device and each application that understands the Internet protocol can participate in the system. IP provides a best-effort delivery system that offers only very little guarantees. Further guarantees can be added on demand. This architecture is often represented by an hourglass model (see Figure 5.1) [158].

On the top of the hourglass, there are the applications that can be used in the Internet. They are very heterogeneous in nature and require different guarantees. Emails, for example, require a reliable transmission whereas streaming applications need the transmission to be as fast as possible while packet losses are accepted. All these applications use the unreliable Internet protocol as foundation. Guarantees that go beyond that need to be added on a higher layer. At the bottom of the hourglass, there are the different physical transmission channels. Protocols on higher layers allow these mediums to understand IP. As a result, all the different applications can communicate via the various transmission channels with each other in a global network that is centered around a minimal abstraction of communication, the Internet protocol.

The Tasklet system borrows from the idea of the Internet protocol. With diverse applications on the one side and heterogeneous physical resources on the other side, Tasklets represent the center piece of the architecture. Tasklets are executed in a best-effort manner and, thus, are lightweight in nature. Applications can add guarantees on top of this best-effort execution in a modular way. That means that they can select one or multiple service guarantees depending on their requirements. This approach has two advantages. First, regardless of their requirements, all applications can use Tasklets as an abstraction for computation. Applications that require multiple guarantees request these guarantees on top of Tasklets, whereas highly-responsive, lightweight applications just use the Tasklet abstraction as is. Second, the modular approach allows to select exactly those guarantees that are required and, thus, add only that respective overhead. This is different to IP, where developers have only two options. They can either chose the lightweight, unreliable option, the User Datagram Protocol (UDP), or the Transmission Control Protocol (TCP) that provides a guaranteed and ordered delivery as well as flow-control. It is not possible to select either a reliable or an ordered delivery. This adds additional overhead for features that are not required. In practice, for IP, this problem can be eventually solved by using UDP and implementing the required guarantees manually in the application layer. To avoid this, we provide modular access to the guarantees in the Tasklet system. To integrate all kinds of heterogeneous devices such as powerful desktop computers and thin mobile devices into the Tasklet system, we provide a lightweight execution environment. Each device that runs this environment can participate as a resource provider in the Tasklet system. In the remainder of this section, we introduce the Tasklet system in detail.

### 5.1.1. System Model

Distributed computing systems allow applications to make use of numerous remote resources. Typically, these systems support a particular application domain. BOINC [9], for example, offers a middleware to distribute the workload of long-lasting scientific computations to personal computers. MAUI [68] performs fine-grained code offloading to remote servers to save energy on mobile devices. Google App Engine (GAE) [159] provides a platform to host web applications that can be accessed without further setup. The reliability of BOINC, the granularity of MAUI, and the flexibility and spontaneity of GAE are inherent in their respective system architecture. In contrast, the goal of the Tasklet system is to provide an easy to use abstraction for developers to distribute the workload of computationally intensive applications to local and remote resources. This abstraction allows devices to execute all different types of applications within all kinds of computation environments, including stable cloud settings or unreliable edge environments.

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

The Tasklet system consists of three entities: providers, consumers, and brokers. Resource *providers* offer their computing resources in form of Tasklet virtual machines (TVMs). Resource *consumers* distribute the workload of computationally intensive applications to providers. Resource *brokers* act as resource controllers and perform the matchmaking between consumers and providers. Figure 5.2 shows the system model of the Tasklet environment. The scheduling of Tasklets works as follows. First, providers register their resources at a broker. A consumer that wants to execute a Tasklet remotely sends an execution request to the broker. The broker selects a provider for execution and returns this information to the consumer. The consumer directly sends the Tasklet to the provider which executes the Tasklet on one of its local TVMs. When the Tasklet is executed successfully, the provider directly returns the result to the consumer.

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

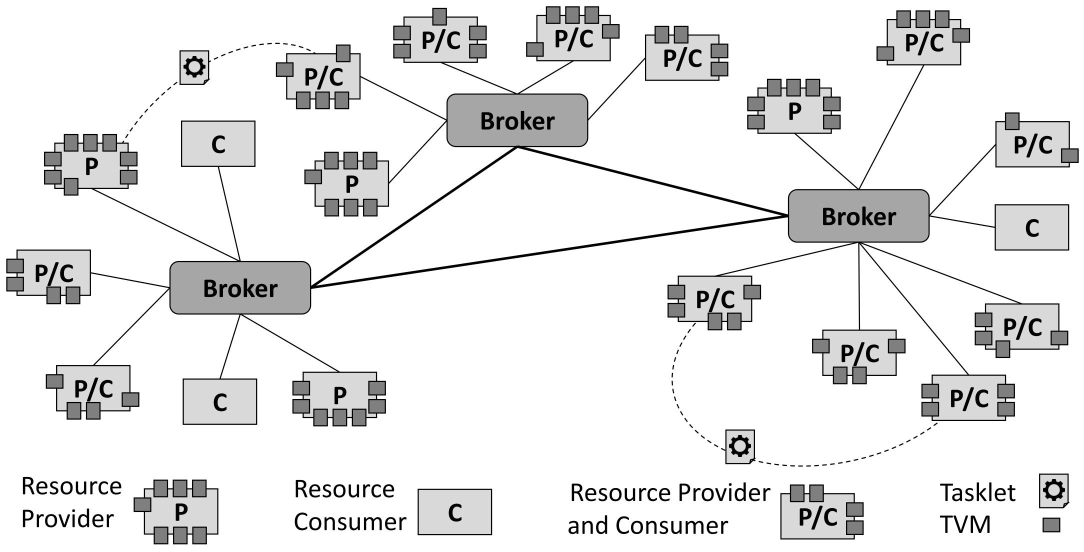


Figure 5.2.: System model of the Tasklet environment. Resource providers (P) share their idle computing capacities with resource consumers (R) which distribute computational workload in form of so-called *Tasklet*. Brokers perform the resource management as well as the matchmaking between consumers and providers.

in a resource pool increases and cannot be handled by a single broker anymore, new brokers can be spawned. Providers and consumers are then shared among the old and new broker instances.

In general, the Tasklet middleware provides a best-effort execution layer. The scheduling and execution of Tasklets is implemented without providing any further guarantees. As a result, Tasklets can be dropped at any point in time and the matchmaking between Tasklets and providers is performed randomly. While this service is sufficient for some applications, other applications might require further guarantees. Therefore, we introduce the concept of *Quality of Computation (QoC)* in Section 5.1.5. QoC adds another layer on top of the best-effort Tasklet system. Application developers can specify QoC goals for Tasklets, such as a reliable execution. The Tasklet middleware then enforces these goals. It monitors the execution and re-initiates it in case the provider crashes during execution. Developers can use QoC goals to affect the context-aware scheduling decision. As an example, developers can request a *fast* execution. The scheduler then selects powerful providers to reduce the execution time and starts redundant executions in case one provider fails.

### 5.1.2. Use Cases

The system model presented above shows the general structure of the Tasklet system. It does not define who the participants in the system are. It shows the technical structure but not the administrative perspective of the system. Thus, it is not specified who is allowed to participate as a resource consumer, who provides the resources, and who runs and maintains the brokers. These questions are left unanswered intentionally, because the Tasklet system is not restricted to a single use case. In the following, we discuss three possible use cases that explain how and by whom the Tasklet system could be used in practice.

#### Global Computing

In our vision, we have hypothetically assumed that Tasklets become the standard unit of computation. All applications are written in a way that their computationally intensive parts are expressed as Tasklets. Thus, each application can use remote resources for execution and all devices become resource consumers. This includes privately owned smartphones and laptops, office computers, smart objects in the Internet of Things (IoT), wearable devices, and also powerful supercomputers that require even more computing cycles than they can provide themselves. At the same time, all devices would run Tasklet virtual machines to execute Tasklets either for local applications or as a service for other devices.

This results in a global system of computation, where all devices collaborate and act as a single system similar to the Internet. The physical borders of single devices would vanish and computational power would turn into a virtualized commodity. The fact that, from a technical perspective, all devices could collaborate does not mean that resource owners are obliged to share their processing power with others. Instead, additional rules and sharing restrictions, social relationships, and accounting mechanisms for compensation mechanisms can be implemented on top of the Tasklet system. Resource consumers and providers could then define with whom they want to exchange resources. Computational resources could also be provided in public institutions, airports, city centers, and other public spaces in form of cloudlets [44]. In such a system, brokers would be provided by central authorities or companies such as the Internet Service Providers (ISPs). They would also maintain the system and provide the necessary infrastructure for the system to reach a global scale.

### Scientific Computing

One domain where the idea of sharing computational resources is already established successfully is the field of volunteer computing for scientific purposes. In particular, BOINC [9] offers a platform for various projects that benefit from device owners who donate the processing power of their devices to the project. This way, the projects can benefit from a large pool of resources that they might not be able to afford if they had to rent resources for example from cloud providers.

The Tasklet system could be used for the purpose of scientific computing as well. When the scientific problems are expressed in form of Tasklets, resource owners can volunteer by providing their resources to the Tasklet system. In this case, the project owners would run and maintain the Tasklet brokers and researchers of this project could use the system as resource consumers. For each scientific project, a separated environment with one or multiple brokers would exist. It would also be possible that multiple projects work together. Resource consumers would not provide their resources for only one project but for a whole group of projects. One or multiple of these projects would be responsible to run the Tasklet brokers.

### Enterprise Computing

A Tasklet environment as shown in Figure 5.2 could also be deployed on a company-wide scale. Large companies typically own hundreds or thousands of computers that are distributed among multiple locations and subsidiaries. The combined computational power of these devices could serve as resources that companies would have to rent from external companies such as cloud providers. The employees of the company would participate in the system as resource providers and at the same time could be resource consumers when they run computationally intensive applications. The company would deploy one or multiple resource brokers and the access to the environment would be limited to computers within the company. This model would benefit globally acting companies in particular. Due to different time zones, employees have different working hours. Hence, for example, computational resources in the United States could be used from Europe when employees in the United States are still asleep.

### 5.1.3. Tasklets

Tasklets are the centerpiece of our distributed computing system. A Tasklet is a closed unit of computation that contains all information, i.e., data and code, which is required to execute the Tasklet and to return the result of the execution to the device which has started the Tasklet. Tasklets are not entire applications, but can contain everything from a few lines of code up to multiple functions that mutually call each other. Application developers can use Tasklets to offload computationally intensive parts of their applications. They do not have to implement the offloading process themselves but use a few API calls to create Tasklets and send them to remote resources. The Tasklet middleware handles the creation of Tasklets, the scheduling, the remote execution, and eventually the result handling transparently for both, application developers and users. By design, Tasklets have three important properties. They are self-contained, lightweight, and independent.

**Self-contained:** Tasklets are self-contained in a way that each resource provider can execute a Tasklet without any additional information. This allows providers to be stateless. They do not have to store any details about the resource consumer or any information about previously executed Tasklets of this consumer. After the execution of a Tasklet, the provider has the same state as before. For consumers this also means that as soon as they have sent out a Tasklet, they can forget about this Tasklet until the result arrives.

**Lightweight:** Tasklets are executed in a best-effort manner. This means that, in the purest form of the system, there are no guarantees about how the Tasklet is executed or whether it is executed at all. The system would not drop Tasklets intentionally without good cause but also would not make any additional effort to guarantee a successful execution. In case a Tasklet gets lost during transmission or when an execution is aborted the consumer would neither receive a result nor get informed about the loss. This makes Tasklets highly lightweight as the consumer does not have to keep any state information. Thus, consumers can start multiple Tasklets at the same time without facing additional administrative overhead. The Tasklet system provides the possibility for applications to request execution requirements. However, this comes at the cost of additional overhead.

**Independent:** Tasklets do not have any side effects. They do not interfere with other Tasklets executed on the same device, do not write on hard drives, and do not interact with each other. This also includes that Tasklets do not interchange intermediate results with each other or share a distributed memory. While this can be seen as a limitation of their functionality, it allows Tasklets to be scheduled and executed without any restrictions or dependencies on other Tasklets. Further, this property makes it possible that a consumer starts a Tasklet multiple times, either to enhance reliability (in case a resource provider fails) or execution time (by taking the first of the results). If an application includes two or more sequential computational tasks, it has to wait for the result of the previous Tasklet first.

### 5.1.4. Tasklet Middleware

The Tasklet middleware runs on resource consumers and providers. It contains all the logic to create a Tasklet, exchange them between devices, execute them, and finally return the results back to the host application. The middleware is designed in a layered architecture with three layers. The construction layer is responsible to create Tasklets for applications. It hands the Tasklet over to the distribution layer which performs the scheduling of Tasklets. When a Tasklet arrives at a resource provider, the execution layer executes the Tasklet and hands the result to the distribution layer. The result gets forwarded to the host device, which is the device that has started the Tasklet in the first place. When the Tasklet result arrives at the host device it gets forwarded to the user application.

The layered architecture of the Tasklet middleware with its components is shown in Figure 5.3. The architecture shows that resource consumers do not need the execution layer and resource providers do not require the construction layer of the middleware. A device that acts as both, provider and consumer, contains all three layers. In the following, we will discuss each layer and its components in detail.

#### Tasklet Construction

Application developers use Tasklets to offload computation to remote resources. Therefore, they create Tasklets within their applications and hand them over to the Tasklet middleware which then takes over the control and performs the scheduling and the execution of the Tasklet transparently. One major design guideline in

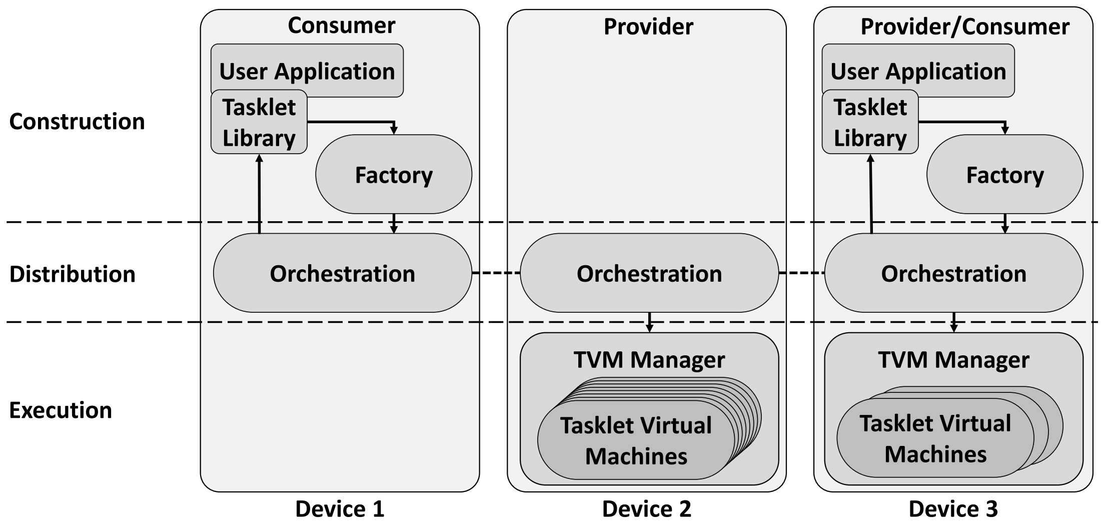


Figure 5.3.: Layered architecture of the Tasklet middleware.

the development of the Tasklet system was the ease of use for developers. Thus, the system allows developers to write applications in their favorite programming language, which we will call host language. Only those parts of the resource-hungry parts of the applications that need to be offloaded have to be written in the Tasklet language, called C--, that has been developed for this purpose. To create a Tasklet in the host language, developers use the Tasklet library which provides an API to create and start Tasklets as well as to receive Tasklet results. Thus, the library connects the application written in the host language on the one side and the Tasklet on the other side. Figure 5.4 illustrates the Tasklet construction process.

In the long term, there is a library for each programming language. So far, Java, Android, and C# are supported. At runtime, when a Tasklet is created, the Tasklet library passes all information to the Tasklet factory. The factory compiles the Tasklet source code into byte code and adds meta data to the Tasklet. The meta data include an application ID, a Tasklet ID, the IP address of the host device, and further information to uniquely identify a Tasklet. The factory creates a closure and sends it to the orchestration module in the distribution layer via a socket. We will now describe the different concepts and modules of the Tasklet creation process in detail.

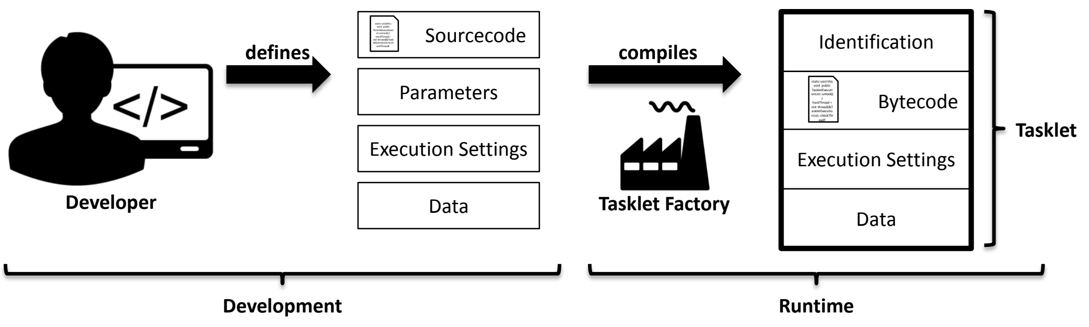


Figure 5.4.: Tasklet construction process

**Host Language Concept:** Application developers can use their favorite programming language for the development of applications that use Tasklets. Typically, applications require a user interface, some application logic, input and output, but also computationally intensive parts. Despite the latter, neither of these parts need to be changed in the Tasklet system. Developers select only those parts of the application that require the most processing power and implement them in the Tasklet language C--. Within the application, Tasklets are created and started by calling methods of the Tasklet library that provides an API for Tasklet handling. This concept has several advantages. First, developers can use their favorite programming languages for most parts of their applications. They can use all concepts and features of these languages such as object orientation and event handling and can also use third-party libraries. Second, existing applications require only minimal changes to offload workload in the form of Tasklets. Tasklets can easily be integrated into existing applications by only replacing the computationally intensive parts. Third, the concept allows for a clear separation of responsibilities. While application developers are responsible for the Tasklet logic, that is what the Tasklet is computing, the Tasklet middleware handles the offloading and execution of Tasklets as well as collecting the Tasklets' results.

Before implementing the host language concept, we discussed several alternatives. One option was to create an own complete language that developers could use to program their entire application. This approach had the advantage that the partitioning of the application could be performed more dynamically. However, it would also imply that this language needed to support all features of a mature programming language such as Python, Java, or C# and that programmers have to learn how to use this language for their entire application.

A second alternative was the usage of an existing language for the definition of the Tasklet code. While this would have been a feasible option, we decided that an own programming language and corresponding execution environment gives us the most flexibility to implement changes in a later point in time, for example automated parallelization.

**Tasklet Library:** The Tasklet library is the connection between the application on the one side and Tasklets on the other side. It provides an API to create Tasklets and to retrieve Tasklet results. Application developers use the API to specify all relevant parts of the Tasklet such as the Tasklet code, the parameters, the QoC requirements, as well as the required data. The Tasklet library adds metadata for identification of the Tasklet, translates all information into a language-independent byte array, and forwards it to the Tasklet factory. In theory, developers could use Tasklets without the Tasklet library and manually translate all information. However, this would result in a tedious and error-prone process.

**Tasklet Structure:** To create a Tasklet, developers have to define four parts. First, they have to write the Tasklet code in C-- that contains functions to be executed by TVMs. The source code is stored in a .cmm-file. When developers use this code, they refer to it via the file identifier. Second, developers have to set the parameters for the source code. In a small example application, where the Tasklet code finds all prime numbers within a given interval, the required parameters would be the upper and the lower bound of the search interval. Third, developers can specify execution requirements, the so-called quality of computation parameters. With these annotations they can request execution guarantees such as a reliable or timely execution. Fourth, developers can add data to the Tasklet.

**Tasklet Language C--:** Tasklets are defined in the Tasklet language C--, that has been developed for this purpose. C-- is a procedural programming language and contains a small set of available functions. The name suggests its resemblance to the programming language C. The decrementer (--) indicates that, in contrast to C++, the language only contains a subset of the functionality of C. Available concepts include primitive data types, loops, conditional statements, and functions. The code can require zero, one, or multiple parameters and produce an arbitrary amount of results. When a Tasklet is created the Tasklet factory compiles the code and the parameters into static byte code.

**Tasklet Factory:** At runtime, the Tasklet factory assembles the Tasklets. Besides the compilation of the source code and the parameters into byte code, it adds the nonfunctional requirements and the data. It packs all parts of the Tasklet into a closed unit according to the format that is well understood by the Tasklet middleware. It adds a header that defines the lengths of the code, QoC parameters, and the data and forwards the Tasklet to the orchestration in the distribution layer.

### Tasklet Distribution

The orchestration module performs the scheduling of Tasklets. It first decides whether the Tasklet should be offloaded or executed locally. A local execution can be required when the device is not connected to the network or when the developer has explicitly forbidden a remote execution, for example, for privacy reasons. If the Tasklet can be offloaded, the orchestration contacts the resource broker and sends a resource request. As the broker has global knowledge of the available resources, it can select the most suitable provider for each Tasklet. The resource request from the orchestration can include further information, such as requirements about the execution speed of the provider. The broker considers these requirements in the provider selection and sends information about the selected provider back to orchestration module at the consumer. The consumer sends the Tasklet to the provider where it is scheduled for execution.

**Orchestration:** The orchestration handles resource requests and the communication between consumers and providers. Tasklets and Tasklet results are exchanged directly between consumers and providers. Once a resource request has been successful and the consumer knows which provider has been selected for Tasklet execution, it directly forwards the Tasklet to this provider. As Tasklets are stateless, the consumer does not store any information about this Tasklet and the provider. Also, the provider does not store any information about the consumer and, thus, the connection between the two entities is closed. Only after the provider has finished the execution of the Tasklet, it sends the results to the consumer. Therefore, it gets all required information about the origin of the Tasklet from the metadata that are stored in the Tasklet closure.

**Tasklet Broker:** Tasklet brokers perform the resource allocation, that is, the matchmaking between resource consumers and providers. Brokers have global knowledge, as providers register at their nearest broker. Therefore, they ping all known brokers and connect to the one with the lowest latency. The broker adds them to its resource list which is continuously updated. Providers send heartbeats to their broker to signal that they are up and running. They also register their number of idle TVMs. Thus, the broker knows where to schedule Tasklets without overloading single providers. Once the broker has selected a provider for execution, it decrements the number of idle TVMs for this provider. When the provider has finished the execution and the TVMs become idle again, the provider notifies the broker which increments the number of idle machines for this provider.

**Resource Selection:** In the most simple case, brokers randomly allocate resources to Tasklets. Each Tasklet has the same chance to get assigned to a fast, slow, reliable, or unreliable resource provider. Resource selection, however, becomes more complex when application requirements are considered. The Tasklet brokers support context-aware task scheduling which will be discussed throughout the remainder of this thesis.

### Tasklet Execution

Resource providers run the execution environment for Tasklets, the TVMs. A provider can run one or multiple TVMs at the same time. The Tasklet Virtual Machine Manager orchestrates the TVMs locally. It is the interface to the distribution layer and schedules incoming Tasklets to idle TVMs. It is also responsible for starting and stopping TVMs and for monitoring the status of the available TVMs. Details of the Tasklet execution layer are discussed in the following.

**Tasklet Virtual Machine:** The TVM is a stack-based byte code interpreter. It is built to be lightweight with a minimal memory footprint, in order to run on many devices, including very small ones, for example, in embedded systems or sensor networks. TVMs are single threaded processes and do not support multi-threading themselves. They sequentially execute incoming Tasklets without interruptions, similar to batch scheduling. However, the resource owner may terminate a TVM at any time, for example, in the excess capacity scenario. TVMs do not support system calls or access to any system resources from the Tasklet code. During execution, a Tasklet's code, parameters and data never leave the volatile memory of the physical machine. Error handling in the best-effort TVM is straightforward. If a run-time error occurs, the execution is terminated and the Tasklet is dropped. Finally, the TVM resets itself to its initial state after each Tasklet execution.

**Heartbeats:** In general, Tasklets are executed in a best-effort manner. This means that if an execution fails, no countermeasures are taken. For applications that do rely on a successful execution of Tasklets, the system provides the option to request a reliable execution. Therefore, the resource provider sends heartbeats to the resource consumer, which maintains a list of all Tasklets that are currently executed remotely. When the consumer detects that a provider has stopped sending heartbeats, it starts the execution of the respective Tasklet again.

### 5.1.5. Quality of Computation

We have introduced the Tasklet system as a lightweight but unreliable task offloading system that does not provide any guarantees for the execution of Tasklets. This design decision imposes the least requirements on computational resources and allows even highly unreliable and volatile resources to be used as targets for computation offloading. Volatile resources can be, for example, excess capacities in local computers or cloud instances that are not used for some time. While these idle CPU cycles could be leveraged by a distributed computation system, they might drop the execution of Tasklets as soon as the resources are required for local processes. In this case, the Tasklet executions are lost. While this best-effort computing is sufficient for some applications, others require a reliable or timely execution.

We introduce the concept of Quality of Computation (QoC) that borrows from the idea of the layered Internet protocol stack where an unreliable lower layer provides the basic functionality of the system. Any further guarantees might be added on a higher layer while leaving the underlying system (mostly) untouched. In a similar way, we implement QoC into the Tasklet system and let different applications use various kinds of guarantees on top of the best-effort execution layer. This design has two advantages. First, it does not introduce any overhead to applications that do not require any guarantees and still allows the lightweight execution of Tasklets. Second, the system is extensible and further guarantees might be added to the system later. We leave the decision whether an application requires any additional execution guarantees up to the developers who most likely have the required domain knowledge and can decide for their individual applications. QoC requirements can be defined on a per-Tasklet level which allows for a different treatment of Tasklets even within one application.

### QoC Design

The fine-granular definition of application requirements makes the Tasklet system usable for multiple kinds of applications. However, not only the applications and their execution requirements are heterogeneous but also the environments in which Tasklets are executed. Environments can consist of fast and reliable resource providers such as static and centrally-administered cloud instances that inherently provide a timely and very reliable Tasklet execution. Resource providers could also be user-owned edge devices or cloud excess capacities that do not only vary in their computational power but might also terminate the execution of a Tasklet at any time. Finally, the execution environment can consist of any combination of these reliable and unreliable resources.

As applications can be executed in such different environments, the enforcement of the execution guarantees cannot be static but must be adapted to the current execution context. In a stable environment with dedicated cloud resources, Tasklets are executed in a reliable manner without the need of further measures. In edge environments with end-user administered devices, executions can fail at any time. Redundant executions of the same Tasklet in parallel can increase the chance of a successful execution. The level of redundancy depends on the reliability of the resources. To guarantee that a Tasklet is eventually executed even in unreliable environments, the system needs to monitor the state of the execution and re-initiate the Tasklet in case of a fault.

This situation with different application requirements on the one side and different types of environments on the other side creates a dilemma. Application developers typically cannot know in which environment their applications are executed and, therefore, cannot determine the optimal way how to enforce QoC guarantees in each environment. The Tasklet middleware, which could measure the environment context and enforce the guarantees depending on this context, however, cannot know which guarantees are relevant for each application. To solve this dilemma,

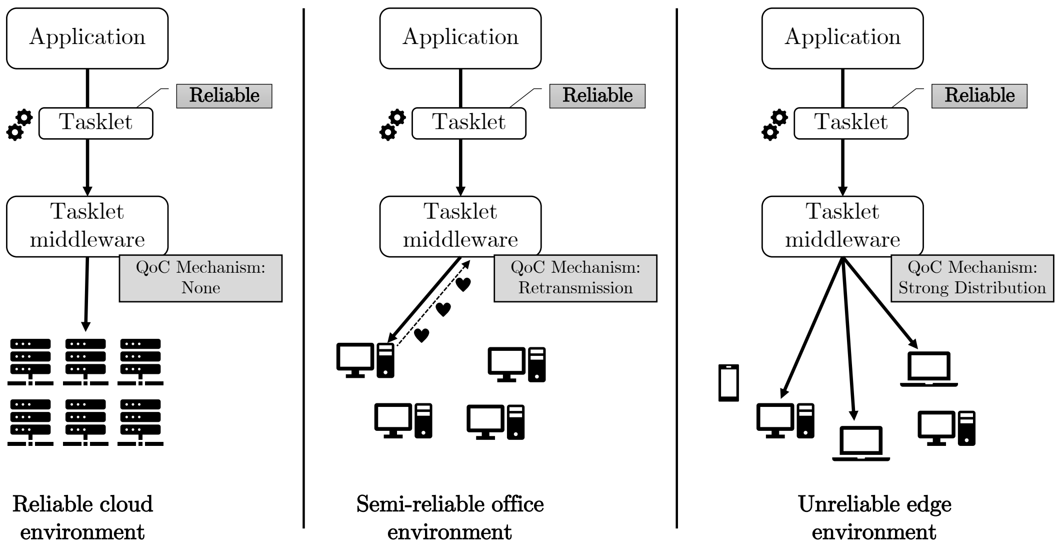


Figure 5.5.: QoC enforcement on the example of the QoC goal *reliability*. For each execution environment, the Tasklet middleware selects a mechanism or a combination of mechanisms to enforce the guarantees for each Tasklet execution.

we decouple the definition of QoC guarantees with the actual enforcement of these guarantees at runtime.

Therefore, we introduce QoC goals and QoC mechanisms. QoC goals are high-level requirements that developers can specify for each Tasklet individually. Examples of QoC goals are *reliability*, *speed*, or *cost*. These goals are enforced by QoC mechanisms such as *retransmission*, *local execution*, or *speed filter*. The Tasklet middleware checks the QoC goals for each individual Tasklet and selects the best mechanism or combination of mechanisms based on the current execution context.

Figure 5.5 demonstrates the QoC enforcement on the example of the QoC goal *reliability* in three different execution environments. In a reliable cloud environment, the Tasklet middleware does not apply any QoC mechanism as the resources are stable and reliable. In an office environment, in which devices typically run for multiple hours without being turned off, the Tasklet middleware uses the QoC mechanism *retransmission* that establishes a heartbeat channel to monitor the Tasklet execution. In case of a failure, the Tasklet is re-initiated. Finally, in a highly unreliable edge environment, the middleware deploys the *strong distribution* QoC mechanism that schedules the same Tasklet to multiple resource providers in parallel. The execution is successful, if at least one Tasklet is executed completely. The degree of redundancy depends on the reliability of the underlying computational resources. In contrast to the *retransmission* mechanism, parallel execution might lead to faster results because in case of a failure, the execution does not need to start from the beginning. The two mechanisms *retransmission* and *strong distribution* can also be combined to give stronger reliability guarantees.

QoC enforcement cannot only depend on the current execution environment but might also take other factors into account, such as user preferences, the context of the local device, or the application state. Here, we do not take these factors into account. However, the Tasklet API allows developers to consider these factors on application level and set the QoC goals for each Tasklet accordingly.

In the following, we discuss multiple QoC goals and QoC mechanisms. These listings are not meant to be exhaustive but can be extended at any time. At the end of this section, the mapping between QoC goals and mechanisms in Table 5.1 shows possible strategies to enforce each QoC goal by one or multiple QoC mechanisms.

### QoC Goals

The scenarios described above show the need of mechanisms to make a wide range of applications usable in a best-effort distributed computing system. Here, we outline the goals that our QoC concept provides.

**Reliability:** In unstructured distributed computation environments, execution is, by itself, unreliable. For some applications, it is sufficient if the execution succeeds with a certain likelihood. The system then executes the task with a given probability of success. Other applications require a guaranteed execution. In this case, the system assures that tasks will eventually be executed. If there are no remote resources accessible, a local execution can serve as a fallback option. Developers can choose from two levels of reliability, *'likely'* and *'guaranteed'*. For *likely* executions, they can set a probability with which a Tasklet will be executed successfully. For *guaranteed* executions, the system keeps trying to execute the Tasklet until, eventually, one attempt succeeds.

**Speed:** Developers can request a timely execution. The system does not provide real time guarantees but selects the best strategy for a fast execution. The actual execution time largely depends on the environment, i.e., the available resources and the network connectivity. The speed goal by itself does not guarantee a reliable execution. However, if the task is executed, it is done in the fastest possible way.

**Precision:** The *precision* goal supports capabilities to enhance the execution accuracy. Multiple executions of tasks can be beneficial in domains of simulation, artificial intelligence, or brute-force algorithms. The idea of the *precision* QoC goal is to execute multiple identical tasks with as little overhead as possible to get a higher number of results. Our system allows the developer to start a high number of computations with a single call. This hides the complexity of Tasklet distribution and result handling. Developers can specify a deadline for the results of a repetitive task. The remote execution produces a series of results until it is interrupted. The interruption happens just in time to ensure a timely result delivery to the application. At application level, developers can then aggregate the results.

**Privacy:** Tasklets might contain sensitive data or confidential algorithms. To avoid disclosure of data or code, developers can set the *privacy* goal for Tasklets which ensures a secure transmission and a trusted host for execution. Trusted hosts are resource providers that are either verified by a central authority or trusted by the user.

**Cost:** Computation often does not come for free but is accounted according to an underlying economic model. Developers can specify that a task is executed in the least expensive way. Though this might impact the speed or the reliability of the execution, the costs will be kept at a minimum. Requesting a fast and cheap execution at the same time results in a trade-off between both parameters. The fine tuning between these goals requires knowledge about user preferences and is beyond the scope of this thesis.

**Energy:** Devices with limited battery capacities can benefit from energy-aware computing. Depending on the context of the device, tasks can be postponed for later execution or even dropped in case the battery is in a critical state.

### QoC Mechanisms

Next, we present the set of QoC mechanisms that can be used to enforce QoC goals. QoC mechanisms are implemented throughout all components in the Tasklet middleware and affect the scheduling of Tasklets as well as their execution. Some mechanisms guarantee a certain behavior, others optimize the execution quality in a particular dimension. The following mechanisms are implemented in the Tasklet system:

**Multiple Execution:** This mechanism issues a set of identical copies of a Tasklet and forwards them to several providers. In case a provider offers more than one resource, the Tasklet is sent once, cloned, and executed several times on different TVMs. The Tasklet is only compiled once and is split up on the orchestration level of the remote instance. Tasklet code and parameters are the same for each copy. For simulations, multiple execution enhances the *precision* of the computation by increasing the number of results.

**Strong Distribution:** Similar to *multiple execution*, a Tasklet is compiled exactly once, but executed multiple times. However, for *strong distribution* each Tasklet is scheduled on a different physical machine to enhance fault-tolerance. For this, the orchestration requests a set of distinct resource providers from the broker. Each instance gets only one copy of the Tasklet. This mechanism supports the *speed* and *reliability* goal to spread the risk of selecting slow, malicious, or unreliable providers.

**Retransmission:** The execution of a Tasklet is monitored in the orchestration by means of a heartbeat channel. After a timeout, the orchestration requests a new resource and re-initiates the execution of the Tasklet. Retransmission enhances *reliability* in a best-effort system.

**Local Execution:** This mechanism enforces a prioritized and instantaneous local execution without sending a resource request to the broker. In case there is no local TVM, the execution is aborted. A local execution is important as a fallback mechanism for *reliability*, in case of a permanent disconnection. Since the Tasklet never leaves the local machine, it provides the highest possible *privacy* level as well.

**Remote Execution:** The Tasklet execution is allowed on every but the local machine. In case of a missing network connection, the Tasklet is dropped. If no *reliability* goal is set, this drop happens silently. Remote execution can preempt exhaustive *energy* consumption of the local device or reserve local resources for more time or privacy critical tasks.

**QoC Mechanisms**

| QoC Goals | Multiple exec. | Strong distr. | Retransm. | Local exec. | Remote exec. | Encryption | Select instance | Speed filter | Timed exec. | Maj. voting | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Reliability | | ✓ | ✓ | ✓ | | | | | | | |
| Speed | | ✓ | | ✓ | ✓ | | | ✓ | | | ✓ |
| Precision | ✓ | | | | | | | | ✓ | ✓ | |
| Privacy | | | | ✓ | | ✓ | ✓ | | | | |
| Cost | | | | ✓ | | | ✓ | | | | |
| Energy | | | | ✓ | ✓ | | ✓ | | | | |

Table 5.1.: Mapping between mechanisms and goals. The QoC layer in the Tasklet middleware selects a single mechanism or a combination of mechanisms to enforce one goal.

**Encryption:** The middleware encrypts the Tasklet before sending it to a remote instance. Results are handled analogously. Encryption enhances *privacy* between the system entities by a secure communication.

**Select Instance:** The orchestration places the Tasklet on a specific instance in the system. Each resource in the system has a globally unique ID. This allows developers to select a single device or a group of devices for the execution. This mechanism can be used to restrict the set of resources to trusted providers and thereby increases the level of *privacy*.

**Speed Filter:** The Tasklet is executed on an instance that is faster than a given value. For that reason, all physical machines that act as resource providers execute a benchmark. The broker selects resources based on their performance to increase the *speed* of the execution. The speed filter itself does not take the network bandwidth into account.

**Timed Execution:** The middleware attempts to execute the Tasklet within a given amount of time to meet a soft deadline. Therefore, it measures the round trip time between the consumer and the provider and estimates the communication delay. It executes the Tasklet for some time until it interrupts the TVM to return the results. All results that have been generated to this point, are sent to the caller. This allows applications to use the available computation time as efficiently as possible. This mechanism can support *precision* for repetitively executed subroutines that produce a series of results.

**Majority Vote:** The orchestration schedules multiple Tasklets in the system similar to the *multiple execution* mechanism. The orchestration collects the results, performs a *majority vote*, and forwards one single result to the application. The complexity of the consensus algorithm highly depends on the type of the results. Thus, the implementation of these consensus algorithms is subject to future work.

**Priority:** Our current implementation of the Tasklet system does not allow for queuing of Tasklets. Tasklets are scheduled only to idle providers and if a selected provider requires the resources for local processes, a Tasklet request is rejected. In the future, queuing of Tasklets might be considered. While this allows for more flexibility, it also introduces the risk that Tasklets experience further delays. Therefore, we provide the *priority* mechanism that allows Tasklets to skip the lines and to be executed before any non-prioritized Tasklets to reduce their overall execution time. Increasing the priority of a Tasklet might be subject to a charge.

### 5.1.6. Tasklet Performance

The execution time of Tasklets is the most important performance measure for the Tasklet system. Especially users of responsive applications demand a timely execution. A fast task execution increases the performance of applications and might save valuable time. The execution of a Tasklet in an error-free environment is the sum of three components, scheduling time ($S$), computation time ($C$), and result handling time ($RH$) (see Equation 5.1). The time for Tasklet creation is excluded here as it is typically negligible.

$$\begin{aligned} Execution &= Scheduling + Computation + Result\_Handling \\ E &= S + C + RH \end{aligned} \tag{5.1}$$

**Scheduling Time:** The scheduling time measures how long it takes to offload a Tasklet from the host device to a local or remote resource. In the local case, the scheduling time is close to zero. When the Tasklet is forwarded to a remote resource, the scheduling includes three parts. First, the resource request from the consumer to the broker, second, the resource response from the broker to consumer, third, the Tasklet transmission from the consumer to the provider. The scheduling time depends on the bandwidth between the devices and the size of the offloaded Tasklet.

**Computation Time:** The computation time measures how long it takes for a TVM to compute the Tasklet. This depends on the processing speed of the underlying hardware as well as on the complexity of the Tasklet.

**Result Handling Time:** After the execution is finished, the resource provider sends the result(s) to the consumer. The bandwidth between the devices and the size of the results determine this duration.

The equation above holds under the assumption that Tasklets are executed in an error-free system where all transmissions are successful and providers do not leave the system during Tasklet execution. As this assumption is not realistic, we extend the formula for environments where failures happen and the consumer has to start the execution of a Tasklet anew. Equation 5.2 reflects failures and restarts of Tasklets. Consumers re-initiate the Tasklet execution when they do not receive heartbeats from the provider (anymore). They cannot distinguish whether the transmission has failed or whether the provider has aborted the Tasklet execution. The formula only holds for reliable Tasklets as unreliable Tasklets will not be executed at all in case of a failure.

$$E = S + \sum_{i=1}^{n} (D_i + RS_i + C'_i) + C + RH \tag{5.2}$$

The execution time in an erroneous system is the sum of the execution time in error-free environments ($S + C' + RH$) and the time that has been used for unsuccessful execution attempts. This extra time can be expressed as the sum of the fault detection time ($D$), the rescheduling time ($RS$), and the intermediate computation time ($C'$).

**Detection Time:** When a resource provider executes a Tasklet, it sends heartbeats to the consumer to signal that the execution is still ongoing. In case of a failure, the consumer stops receiving heartbeats and restarts the Tasklet execution. The delay between the occurrence of the fault and the fault detection is called detection time. It depends on the heartbeat interval and the timeout before a task is restarted.

**Rescheduling Time:** A restarted Tasklet also needs to be scheduled to a resource provider. Similar to scheduling, rescheduling includes resource request, response, and Tasklet forwarding.

**Intermediate Computation Time:** When providers abort a Tasklet execution, the progress of this execution is lost. However, the required time adds to the overall Tasklet execution time $E$, as the consumer would not restart the Tasklet while the current execution is still going on. Thus, an aborted execution does not only waste computational resources as the progress of the execution is lost, it also delays the overall Tasklet execution.

In the following, we present multiple improvements for the Tasklet system that have a positive effect on different parts of the execution time. We have implemented and evaluated several of these approaches in the Tasklet system. These will be discussed in more detail in the remainder of this thesis.

### Decentralized Scheduling: S↓, RS↓

Tasklets are scheduled on demand. This means that once a consumer creates a Tasklet, it sends a resource request to the broker and waits for the response. Due to the centralized architecture of the resource allocation, the consumer has to contact the broker to decide where to forward the Tasklet to. This round-trip time adds to the Tasklet execution time and depends on the bandwidth of the connection between the devices. One solution to avoid this delay is to equip each Tasklet with information about available providers and, thus, decentralize the scheduling process. Consumers could then look up a provider from their local list without the need to contact the broker. The exchange of the provider information happens asynchronously and independently from the Tasklet offloading. This reduces the delay in scheduling $S$ and rescheduling $RS$ at the cost of additional overhead for the provider information exchange between broker and consumers. The approach is discussed in detail in Section 6.3.

### Resource Reservation: S↓, RS↓

The consumer performs one resource request for each Tasklet. This is in line with the idea of the independence of Tasklets, whereby each Tasklet is treated in isolation. However, this approach suffers from multiple delays in applications that start multiple Tasklets within a short amount of time. Instead of sending one request per created Tasklet, the system could buffer Tasklets and send a joint resource request to the broker. While this would reduce the number of resource requests, it also introduces delays that might be unacceptable for applications.

A related approach is to make a resource reservation where applications ask for more than one resource per request or the broker predicts the resource demand proactively. The consumer can then store the additional resources and use them for future Tasklets. This approach is in particular useful for applications that create Tasklets on a regular basis. It is discussed in detail in Section 6.4.3 and [160].

#### Fast Heartbeats: D↓

A provider that is executing a Tasklet sends heartbeats to the host consumer to show that the computation is still ongoing. The consumer periodically checks whether Tasklets are outdated and have to be started anew. The delay that is caused by this mechanism depends on three factors: First, the heartbeat interval, that is the time between two subsequent heartbeats, second, the frequency with which the consumer checks for outdated Tasklets, third, the threshold that the consumer uses to define a Tasklet as outdated. The shorter the heartbeat interval is, the higher can be the check frequency, and, as a result, the lower can be the timeout threshold. While this reduces the fault detection time, it adds message and computation overhead.

#### Tasklet Migration: C↓, C'↓, C"↓

In Equation 5.1.6, the computation time $C$ is split up between the intermediate computation time ($C'$) and the final computation time ($C''$). The intermediate computation time is the execution time of a Tasklet before a fault occurs. In general, this time and also the execution progress is lost. By means of Tasklet migration, the execution can be migrated to another provider and the progress can be saved at least partially. The final computation time is the time that is required to execute the rest of the Tasklet starting from the saved progress. Without Tasklet migration, the final computation time $C''$ equals the computation time $C$ from Equation 5.2.

$$E = S + \sum_{i=1}^{n} (D_i + RS_i + C'_i) + C'' + RH$$

In Tasklet migration, the provider takes a snapshot of the current state of the TVM and sends it to the consumer. We distinguish between two kinds of migration, proactive and reactive migration. Proactive migration periodically sends snapshots of the current state. This allows to handle implicit leaves, that occur when providers leave the system or abort the execution without prior notice. In this case, only the progress since the last snapshot is lost and it holds that $C \leq \sum_{i=1}^{n} (C'_i) + C''$. When devices leave the system explicitly, they perform a reactive migration. Before they leave, they take a snapshot of the current TVM state and send this snapshot to the consumer. As no computation is lost, it holds that $C = \sum_{i=1}^{n} (C'_i) + C''$. Proactive migration comes at the cost of snapshot management and transmission which leads to a tradeoff between performance and overhead. Reactive migration causes a small delay as the state has to be sent to the host consumer. The concepts of proactive and reactive migration have been published in [161].

### Speed Filter: C↓, C'↓, C"↓

Resource providers have different processing speeds, which directly determine the execution time. The faster the provider's execution speed, the smaller the resulting computation times $C$, $C'$, and $C''$. The broker can take the processing speed of providers into account when it makes the scheduling decision. Therefore, each provider executes a benchmark Tasklet and sends the results to the broker. While the performance of a provider can vary, for example because of its utilization, these benchmark results give the broker an idea about the execution speed of the providers. The broker then selects the fastest available provider for the execution. The idea of the speed filter is published in [157] and is discussed in Section 6.1.

### Fault-Avoidance: n↓

When a provider aborts the execution of a reliable Tasklet, the consumer has to start the execution anew. These restarts delay the Tasklet execution since fault detection, rescheduling, and computation are performed for each unsuccessful attempt. As a consequence, it is desirable to not only make the Tasklet system fault-tolerant but to try to avoid failures in the first place. As failures cannot be predicted reliably, heuristics and predictions can help to select providers that fail with only a small probability. A scheduler that learns from previous provider behavior can estimate the remaining time a provider might remain online. It can then schedule Tasklets to the most reliable providers and thus minimize the number of retries $n$. The approach is published in [162] and will be discussed in Section 6.2.

### Result Streams: RH↓

When the provider has finished the execution of a Tasklet, it sends the results to the host consumer. Depending on the size of the results, this process might take several seconds up to minutes. To reduce the time for the result handling $RH$, the provider could start sending the results as soon as the first results are available. The provider continuously streams the results to the host consumer where they are buffered. Once all results have arrived at the host consumer, they are delivered to the host application.

### Multiple Execution: E↓

When Tasklets are executed in unreliable environments where failures occur randomly, the execution can be delayed by unsuccessful execution attempts. In addition, slow providers require a longer time for the execution, especially when they are heavily utilized. As these delays cannot be predicted reliably, there is no guarantee for a timely Tasklet execution. However, by starting the same Tasklet multiple times and waiting for the first result, there is a higher chance to retrieve the results in time. The execution time for a Tasklet that gets started $n$ times can then be computed as $E = min(E_1, E_2, E_3, ..., E_n)$ where $E_i$ is the execution time of the $i^{th}$ copy of the same Tasklet. This approach does not accelerate the execution of a single Tasklet directly. However, if at least one copy of the Tasklet gets executed by a fast provider without failures, the host consumer retrieves the result in time. This approach is published in [157] and is discussed in Section 6.1.

### Parallelization: E↓

The Tasklet system helps consumers to make use of a large pool of resources. This has two advantages. First, the consumer can select the most powerful resource providers for the execution. Second, if the structure of the application allows to split up the task into independent parts, the parts can be computed on multiple machines in parallel. The number of parts can exceed the available resources on a single machine by far. In the Tasklet system, an application can create an arbitrary amount of Tasklets to compute a single task. A task, for example, could be an image filtering. This computation could be divided into multiple Tasklets which each compute one part of the image. The execution of the task is done as soon as the last Tasklet has been computed. When the Tasklet is split up into $s$ parts, the task is completed in $T = max(E_1, E_2, E_3, ..., E_s)$ where $E_i$ is the execution time of the i-th Tasklet of the task $T$. Note that this is an optimization on task level and not on Tasklet level.

#### Performance-Aware Scheduling: C↓, C'↓, C"↓

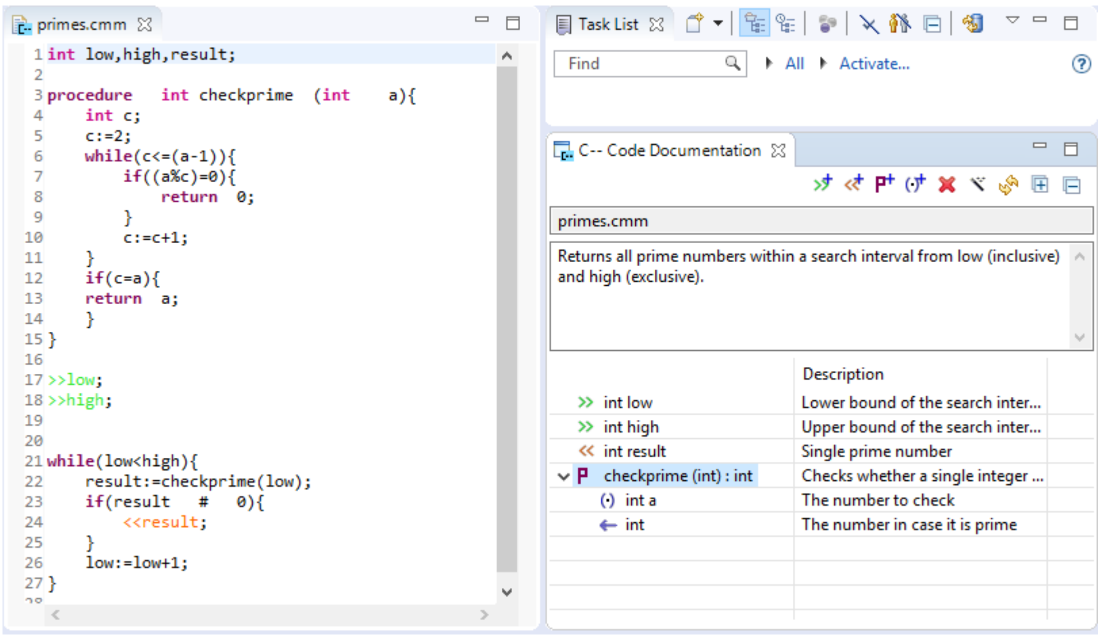
Performance-aware scheduling is another approach on task level. As providers are heterogeneous in their performance, some providers compute Tasklets slower than others and constitute bottlenecks in the system. When task $T$ gets split up into $s$ Tasklets, the slowest provider determines the overall execution time $T = max(E_1, E_2, E_3, ..., E_s)$. To avoid these bottlenecks, the scheduler can make use of the information about the providers' execution speeds from the benchmark measurement. It divides the task not into equal shares but splits it up into shares that reflect the performance of the providers. Fast providers get a larger share, slow providers get a smaller share, respectively. As a result, the variance among the executions is reduced and slow providers are no longer bottlenecks. This approach requires knowledge about the complexity of the task and about how a task can be split up into smaller parts. It is published in [161].

## 5.2. Implementation

This chapter introduces the Tasklet prototype. The prototype is a fully functional distributed middleware. It runs on desktop computers and notebooks with Windows, Mac OS, and Linux operating systems. There is also a version of the middleware that runs on mobile phones with an Android operating system as well as a version for graphics processing units. These versions have minor and major differences in the design and architecture of the middleware, in particular the execution layer. This chapter, unless otherwise stated, refers to the implementation for desktop computers.

### 5.2.1. Tasklet Language C--

The Tasklet logic is written in C--. The language supports the basic programming constructs that are required to write the computationally intensive part of an application. Existing development environments do not support programmers in



```
 1 int low,high,result;
 2
 3 procedure   int checkprime  (int    a){
 4     int c;
 5     c:=2;
 6     while(c<=(a-1)){
 7         if((a%c)=0){
 8             return  0;
 9         }
10         c:=c+1;
11     }
12     if(c=a){
13     return  a;
14     }
15 }
16
17 >>low;
18 >>high;
19
20
21 while(low<high){
22     result:=checkprime(low);
23     if(result   #   0){
24         <<result;
25     }
26     low:=low+1;
27 }
```

primes.cmm

Returns all prime numbers within a search interval from low (inclusive) and high (exclusive).

| | Description |
|---|---|
| >> int low | Lower bound of the search inter... |
| >> int high | Upper bound of the search inter... |
| << int result | Single prime number |
| P checkprime (int) : int | Checks whether a single integer ... |
| (·) int a | The number to check |
| ← int | The number in case it is prime |

Figure 5.6.: Tasklet Code Development Plugin.

writing C-- code which makes the development process error-prone. Thus, we have developed an Eclipse plug-in for C-- implementation. The plug-in provides text-based features such as syntax highlighting, auto indentation, and auto completion. Further, it enhances the documentation of the source code which can be exported to a standalone document. Figure 5.6 shows the plug-in with an example C-- code.

The C-- code on the left side finds all prime numbers within a specified range from `low` to `high`. Global variables are defined in the beginning of the code, followed by the definition of procedures. The only existing procedure in this example is the method `checkprime`, which returns a given number `a` in case this number is prime, zero otherwise. The rest of the code can be compared to the `main(..)` method in Java, as the execution of the code starts from there. In the example, this is line 16.

The >>-operator connects the host language with the C-- code. Each time it is called, it reads the next parameter from the Tasklet into the variable. In this code, two variables, `low` and `high`, are defined. Similarly, the <<-operator writes results into the Tasklet result array that is returned to the host application. The right side of Figure 5.6 shows the documentation features of the plug-in. It automatically detects all input and output variables as well as the procedures. It also provides the possibility to write a documentation for each element as well as the overall C-- code. This allows to share C-- among developers.

### 5.2.2. Tasklet Library

The Tasklet library provides a convenient access to the Tasklet system for developers. It offers an API that allows to create Tasklets and to start their execution with only a few lines of code. The actual process of the Tasklet creation is hidden from developers and does not require manual interaction. The library fulfills three major tasks. First, it provides an API for the definition of Tasklets. Second, it performs the marshalling of the Tasklet components from the host language into a platform-independent bytecode format and forwards the marshalled data to the Tasklet middleware. Third, it buffers the incoming results from the middleware and delivers it to the host application on demand. Tasklet libraries are programming language specific and can be implemented for most programming languages. We have implemented libraries for Java, Android, and C#. In the following, we discuss the features on the example of the Java library. However, libraries in other languages only differ in minor details.

**Tasklet Definition:** Application developers can create and start Tasklets without leaving their favorite programming language. As shown in Figure 5.7, the Tasklet definition follows three simple steps. First, developers create a new object of the type `TaskletBundle`, which can hold one or multiple Tasklets with the same logic ①. The logic, that has previously been defined in a *.cmm* file, is passed as a parameter to the `TaskletBundle` constructor. In the second step, developers add parameters to the Tasklet ②. Therefore, they retrieve a `TaskletParameterList` from the newly created `TaskletBundle` object. The parameter list stores information about the data types and the names of the parameters that the Tasklet logic in the *.cmm* file requires. When developers add parameters line by line, the library checks parameter types and names. In case of a violation, for example when developers misspell the name of a parameter or when they use the wrong data type for a parameter, the library throws an exception at runtime. Each `TaskletParameterList` is then added to the `TaskletBundle`. Adding Tasklets to a `TaskletBundle` instead of starting them directly makes it

```
private static void testPrimes(){

①    TaskletBundle tasklets = TaskletBundle.fromFile("primes.cmm");
   ----------------------------------------------------------------------------
     TaskletParameterList parameters = tasklets.getNewParameterList();
②    parameters.addInt("low", 1);
     parameters.addInt("high", 25000);
     tasklets.addParameterizedRun(parameters);
   ----------------------------------------------------------------------------
③    tasklets.setReliable();
     tasklets.setSpeed();
   ----------------------------------------------------------------------------
④    tasklets.start();
   ----------------------------------------------------------------------------
     TaskletResultPool allResults = tasklets.waitForAllResults();

     for (TaskletResult nextResult : allResults.values()) {
⑤
        for (Object nextPrime : nextResult.getInt(0)) {
           System.out.println("Prime Number: " + (int) nextPrime);
        }
     }
}
```

Figure 5.7.: Tasklet library on the example of a prime number finder.

easy to start multiple Tasklets at once. Developers simply have to repeat step ② to create a new Tasklet with different parameters. In the third step, developers can add QoC goals, such as a reliable and timely execution ③. The API has one method call for each QoC goal and can be extended when new goals are defined.

**Tasklet Launch:** The Tasklet middleware accepts the input for a Tasklet in form of a byte array. This makes the usage of the system platform-independent and lightweight. However, translating the components of a Tasklet into byte code is a tedious and error-prone procedure when performed manually. Thus, the Tasklet library shields the complexity of the translation from the developers who can launch Tasklets with a single line of code. By calling the `start()` method on a `TaskletBundle`, the library translates the *.cmm* source code, the parameters, and the QoC goals into byte code ④. Further, the library adds metadata to the Tasklet, such as a unique identifier, information about the host device and the host application as well as the sizes of code, parameters, and QoCs. In case the Tasklet middleware on the host device is not running yet, the library starts the middleware and forwards the translated byte array to the Tasklet factory where the source code gets compiled into byte code and the final Tasklet is created.

**Result Handling:** Once the application has launched the Tasklets, the middleware handles them without further interaction required. The application can continue running while the Tasklets are executed remotely. The results of executed Tasklets gradually arrive at the middleware of the host device, which forwards them to the host application where they are buffered by the Tasklet library. Developers can access the results by a single line of code with the function `waitForAllResults()` which returns all results from this `TaskletBundle` into an object of the type `TaskletResultPool` ⑤. The method call blocks the application until all Tasklet results have arrived. The results from the individual Tasklets can be retrieved sequentially or via their unique identifier. For applications that do not need to wait for all results but can handle situations where only a subset of Tasklet gets executed, developers can define a timeout. The `waitForAllResults()` method continues after the timeout has occurred and delivers only those results that have arrived by then.

### 5.2.3. Tasklet Protocol

The Tasklet protocol is the communication protocol that the entities in the Tasklet system use to communicate. It defines the structure of the messages as well as the order in which these messages are sent. Messages are directly sent as big endian byte arrays via sockets which allows the communication between different platforms and different programming languages. We provide marshalling and demarshalling methods in C#, Java, and Android to make the protocol more convenient to use.

#### Format

Messages in the Tasklet protocol consist of a protocol header and a message body. The protocol header consists of three integers. The first one is the *magic* number that identifies messages of the protocol and allows to detect messages from other applications and errors during the communication. The second integer defines the protocol version. Versioning allows to incrementally extend the protocol while providing backwards compatibility with previous versions of the protocol. To this date, there are two protocol versions in the Tasklet system. The third integer stores the message type. The message type allows to interpret the remainder of the message. The different message types are discussed below.

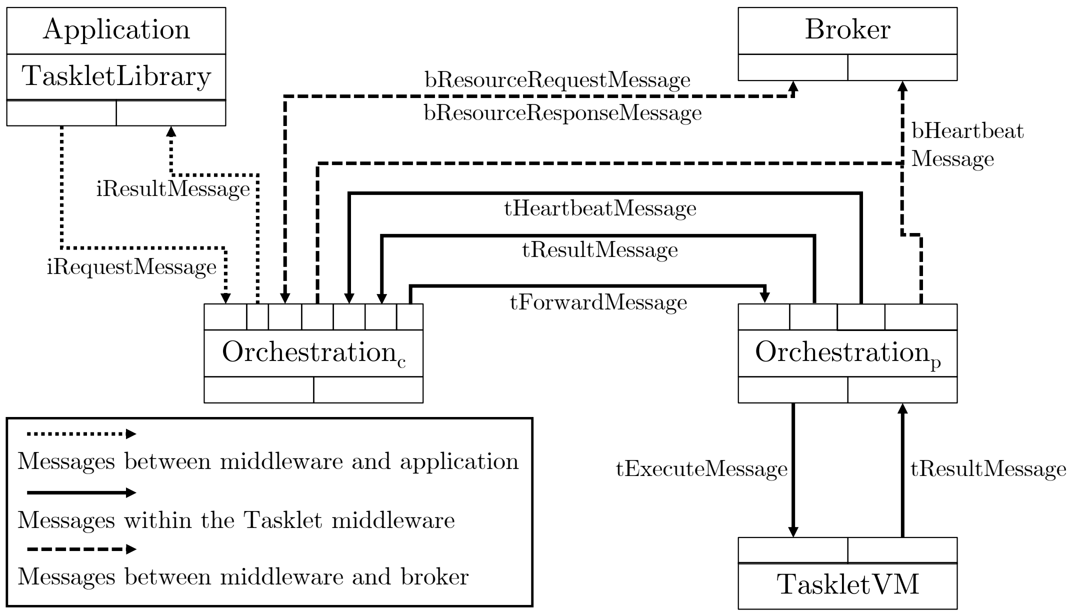


Figure 5.8.: Message flow in the Tasklet protocol. All messages that are sent during a Tasklet execution are shown. For simplicity, the consumer (*Orchestration$_c$*) does not run any TVMs itself and the provider (*Orchestration$_p$*) does not create any Tasklets.

## Messages

There are three types of messages in the Tasklet system. `InterfaceMessages` are used to communicate between the Java Tasklet library and the Tasklet middleware written in C. Instead of Java, any other programming language could be used as well. Resource consumers and providers exchange `TaskletMessages`, for example, to send Tasklets for execution and Tasklet results. Finally, consumers and providers communicate with the broker by means of `BrokerMessages`. Figure 5.8 shows the basic messages in the Tasklet system. In total, we have implemented 35 different message types.

Each message type inside the Tasklet middleware consists of a type-specific header and a payload. The headers include a unique Tasklet identifier that includes the IP address of the local device and the port number that the application uses to receive results from the middleware. The combination of IP address and application port number uniquely identifies an application. The headers further include a Tasklet serial number that is incremented whenever an application creates a Tasklet and a sub-serial number which identifies each individual copy of the same Tasklet.

### 5.2.4. Applicability

In general, all applications can use Tasklets to offload computation. However, some applications can achieve better performance gains than others. Here, we briefly discuss which properties of applications make them better suited for Tasklets than others.

**Computationally Intensive:** As the key benefit of Tasklets is to offload computation to remote resources, an application should contain a sufficient amount of computationally intensive tasks. For only short computations, the overhead involved by the offloading process might require more processing power, time, and battery than a local execution of the task. The Tasklet system does not make the decision whether an offloading decision is beneficial or not but leaves this responsibility to the application programmer. Automated offloading decisions have been discussed, for example, in [68, 163, 164].

**No local dependencies:** In some situations, offloading tasks is no option as these tasks have local dependencies. They might require access to a sensor of the device, local information, or user interaction. As these operations need to be executed on the local device, tasks either have to be returned for this operation or the information has to be sent to remotely executed task. However, this would require dependencies between the host application and Tasklets, which violates the design principle of Tasklets as isolated units of computation.

**Little data dependencies:** The fact that Tasklets are closed units of computation means that everything that is required for execution is packed into the Tasklet. This includes the data for the computation. This data can become large, for example, in video editing applications or big data analytics. The overhead to transfer the Tasklets to remote resources increases accordingly.

**Parallelization:** The Tasklet system can not only increase the performance of applications by running the computation on more powerful devices. It is also able to exploit parallelism and use tens or hundreds of processing units at the same time. Thus, for this period, the host application has access to the resources comparable to a supercomputer and the execution of complex tasks can be increased by multiple orders of magnitudes. This only works when the task can be split up into isolated units of computation that can all be scheduled and executed independently from each other.

**Small results:** Tasklet results have to be returned to the host device. While some computations only return a single integer or a boolean value, other applications produce large result data. Similar to the input data, the overhead increases with the size of the Tasklet results. Having this in mind, developers might consider this in their application design. In general, applications that produce less data are better suited for the Tasklet system.

### 5.2.5. Example Applications

We have implemented multiple applications to show how Tasklets can be used in practice and to evaluate the performance and usability of the approach. In the following, we introduce a subset of these applications. All applications were implemented in Java first. In a second step, the computationally intensive part was replaced by Tasklet code.

**Prime Number Finder:** The prime number finder is a mathematical tool to find prime numbers within a given interval. The application checks each number between the lower and the upper bound of the interval and returns all numbers that are prime. We use this application in multiple examples as it is the almost ideal use case to show the advantages of the Tasklet system. The application requires only very little input data, namely the two integer numbers lower and upper bound. The task consists of multiple independent computations and can be split up into multiple subtasks on arbitrary positions. The amount of result data increases with the problem size but still remains within manageable limits as each subresult is a single integer value. The complexity of the task increases with the size of the interval. Further, the higher numbers, on average, require a longer time to be checked.

**$\pi$ Approximation:** Another mathematical task is $\pi$ approximation where $\pi$ is defined as the ratio of the circumference and the diameter of a circle. Several approaches exist to approximate $\pi$. In the Tasklet system, we use a Monte-Carlo-based approximation. The algorithm creates random points $(a, b) \in (0, 1) \times (0, 1)$ and places them in a coordinate system. In the end, all points that lie within the unit circle are divided by the total number of points $n$. The higher the total number of points the closer the algorithm approximates $\pi$. In contrast to the prime number finder, this algorithm can make use of best-effort resources where failure might occur and the Tasklet execution might not be successful. If $n = 10,000$, the problem can be split up into 10 Tasklets, each of which computes 1000 points. The results are merged in the host application. If one Tasklet is not executed, $n$ is decreased to $9,000$ but the algorithm would still work correctly.

**Matrix Multiplication:** The complexity of matrix multiplication increases with $n^3$ with the dimension of the matrices. The task can be split up into multiple independent subtasks and is therefore well suited for parallel executions. The complexity of the computation is evenly distributed which allows to create Tasklets with almost uniform complexities.

**Mandelbrot Set:** Another application is a graphical representation of the Mandelbrot set, which is a set of complex numbers $c$ for which the function $f_c(z) = z^2 + c$ does not diverge for $z$ iterated from 0. Each frame of the width $w$ and the height $h$ consists of $w * h$ pixels that all have to be tested for divergence. When mapped to a color range, the numbers result in a recognizable pattern. The problem can be split up into multiple subtasks where the subtasks may vary significantly in their complexity as different parts of the set require much more computation than others.

**Connect Four Artificial Intelligence:** The goal of this two-player game is to drop coins into the columns of a $7 \times 6$ matrix, such that four coins of the same player are connected horizontally, vertically, or diagonally. The artificial intelligence is based on a Monte-Carlo simulation and simulates as many sample games as possible without knowing any game strategy. Based on the randomized simulations, the AI selects the move with the highest potential to win. The algorithm stands representative for applications that benefit from multiple executions and can cope with a best-effort service. For these types of algorithms, a higher number of simulations, on average, results in a more accurate prediction. At the application level, all Tasklet results that arrive before a timeout of one second are aggregated and the application determines the next move.

**Option Pricing:** A further application of Monte-Carlo simulation is option pricing. As the computation of option prices is very complex, simulations help to approximate the prices. According to the law of large numbers, the accuracy of the approximation increases with the number of simulation runs. Similar to the $\pi$

approximation, the option pricing application only requires best-effort resources. The total number of simulations can be distributed among multiple Tasklets.

**Ray Tracing:** We have also implemented an application that renders photorealistic images by using ray tracing. In this technique, developers can mathematically define a three-dimensional model of a scene and add lighting and perspective details. The image is generated by tracing the path of light for each pixel in an image plane and can simulate reflection, scattering, and chromatic aberration. As the path of each pixel can be computed independently developers can split up the workload into multiple Tasklets.

**Ant Colony Optimization:** One application uses a nature-inspired heuristic to compute the shortest path for the Traveling Salesman Problem. In the use case for the ant colony optimization, we have implemented different strategies for this metaheuristic with Tasklets. These different strategies differ in the structure of the algorithm. Some of the round-based approaches need a central coordination where all Tasklet results have to be collected by the host application before the next round of Tasklets can be started. This use case shows how the structure of the algorithm can be considered to optimize the performance of the offloading process.

**k-Means Clustering:** As an example of machine learning algorithms we have implemented k-means clustering. The algorithm solves the problem to partition $n$ observations into $k$ clusters and is NP-hard. In each iteration, the distance from all data points to all cluster means is computed and new cluster means are selected. As for each iteration the results have to be aggregated, the application starts a new set of Tasklets for each round.

## 5.3. Evaluation of the Prototype

We evaluated the prototype of the Tasklet system in a brief case study to demonstrate that the system works in real world environments. The focus of the evaluation was threefold. First, we ran the system on multiple types of devices and operating systems to show that it is independent from the underlying platform. Second, we showed the system's capabilities to schedule and execute parallel tasks.

| Device Type | CPU | Single Threaded | Fastest |
|---|---|---|---|
| Office Computer (Intel Q6600) | 4 x 2.4 GHz | 309s | 85s |
| Office Computer (Intel i7-4770) | 8 x 3.4 GHz | 120s | 26s |
| Office Notebook (Intel i7-3520M) | 4 x 2.9 GHz | 331s | 90 |
| Amazon EC2 - c4.8xlarge | 36 x 2.9 GHz | 282s | 9s |
| 10 x Amazon EC2 - c4.8xlarge | 360 x 2.9 GHz | 282s | 2s |

Table 5.2.: Resources used for prototype evaluation. We ran the MBS computation on each device in a single thread. Afterwards we increased the level of parallelism to decrease the computation time. The results of the single-threaded execution and the fastest parallel execution are indicated.

Third, we offloaded Tasklets to test whether the system facilitates the remote execution of multiple parallel tasks on remote devices.

### Use Case

We used the rendering of a Mandelbrot set (MBS) as the sample application to evaluate the prototype. The MBS application is well-suited as a test candidate as it requires only very little input but results in a high computational workload. Further, the application can easily be split up into smaller subproblems that can be executed independently. For the evaluation we repeatedly computed a $640 \times 480$ pixel MBS. As the complexity of the task changes with the selected part of the MBS, we always computed the same section to keep results comparable. During the evaluation, more than 2,000,000 Tasklets were executed in about $1,400$ CPU hours.

### Device Support

We ran the Tasklet system on multiple types of devices, including static computers and laptops running various Windows and Linux operating system versions. We also deployed the Tasklet middleware on several smartphones and tablets running the Android operating system from version 2.3.1 to 6.0. Further, we tested the system on an Amazon Kindle Fire HD with FireOS. Finally, we ran pretests on an Intel Xeon Phi coprocessor, a GPU, and a Raspberry Pi. These tests have shown that the Tasklet system can be deployed on the major devices and platforms and that Tasklets can be shared among all these different computing devices.

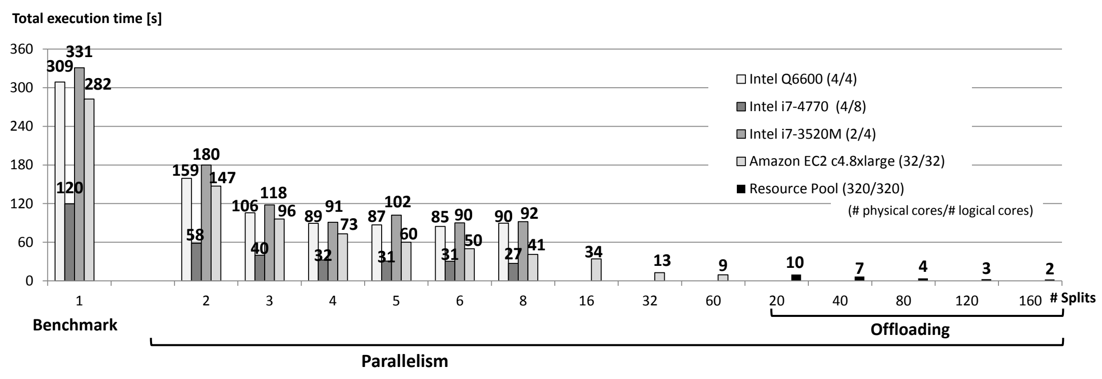


Figure 5.9.: Benchmark of the Tasklet system on the example of the computation of a 640 x 480 pixel Mandelbrot set. The benchmark on the left shows the total execution time of the local computation on four devices (three physical machines and one EC2 virtual machine). The same task is then split up into 2 to 8 parts for the physical and 2 to 60 splits for the virtual machine. In a second step, we offloaded the work to a resource pool of 10 Amazon EC2 instances and used an even higher level or parallelism with up to 160 splits. This accelerated the execution further.

## Benchmark

To establish a benchmark for our system, we analyzed the execution of one MBS computation on a single TVM. We repeated this measurement for a variety of systems (compare Table 5.2). Static parts of the application such as assembling the MBS image from the Tasklet results are excluded from the execution time. The average results of 10 runs range from 331 seconds on the slowest provider to 128 seconds on the fastest machine. They can be found in Figure 5.9 (Benchmark).

The lifecycle of a Tasklet can be divided into three phases. The creation ($T_{cr}$) comprises the compilation and assembling process. Scheduling ($T_{sch}$) is the sum of requesting a resource for execution, forwarding the Tasklet to a host, and returning the result. Interpretation ($T_{int}$) measures the time the Tasklet is interpreted by a TVM. Thus, the time that a Tasklet is handled by the components in the middleware can be formally expressed as: $T_{mw} = T_{cr} + T_{sch} + T_{int}$. Additionally, the application measures the total execution time $T_{ex}$, which is the time between sending the Tasklet request and receiving the corresponding result. Thus, we are able to calculate the network delay as follows: $T_{delay} = T_{ex} - T_{mw}$.

In case of the local benchmark, $T_{ex} \approx T_{int}$, since no network delay is involved and the creation and local scheduling of a single Tasklet happen almost instantly.

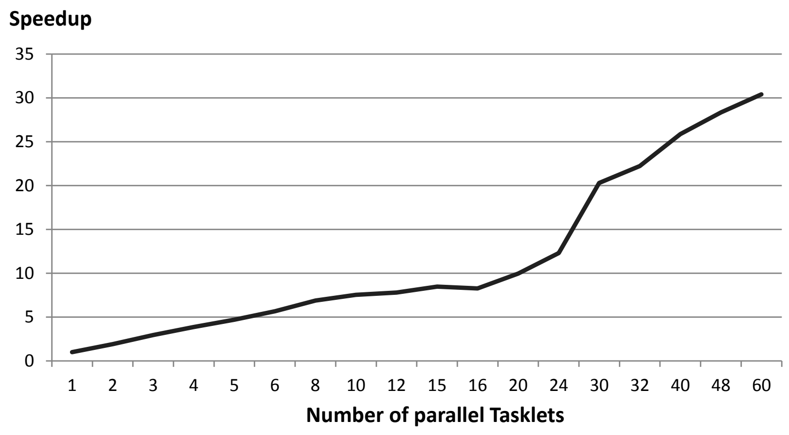


Figure 5.10.: Speedup of the MBS task through parallelism and offloading to cloud resources. The results show that parallel local executions of a high number of Tasklets can be efficiently performed by the Tasklet system.

### Parallelism

Since modern computing systems have multiple CPUs, splitting up the workload on a local device might result in a faster execution. We evaluate, how well the Tasklet middleware is suited to support parallel computation. The Tasklet middleware does not offer automated parallelization mechanisms. However, it provides a convenient way for the programmer to issue multiple parametrized Tasklets. Figure 5.9 (*Parallelism* $2-60$) shows the results of the parallel executions.

In this benchmark, the interpretation time $T_{int}$ still accounts for the majority ($> 95\%$) of the total execution time $T_{ex}$. The performance increase of parallelization on a local device is limited to the number of cores.

### Offloading

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 our applications, we further split up the MBS calculation and run the same task in an even finer granularity between 20 to 160 single Tasklets. For execution, we exclusively used remote resources. Again, each Tasklet had to be compiled, scheduled, and executed and the results had to be sent back to the application. Figure 5.9 (*Offloading* $20 - 160$) shows that offloading can significantly benefit our application.

### Conclusion

The evaluation of the prototype suggests that the Tasklet system works as intended. It can be used on multiple heterogeneous platforms and allows the efficient scheduling on local and remote resources. Figure 5.10 shows the speedup of the MBS task execution on a single Amazon EC2 c4.8xlarge cloud instance. The Tasklet system introduces only a minimal overhead. In the highest level of parallelism, the task is split up into 60 independent Tasklets which results in a speedup of more than 30 times compared to the single-threaded execution on this machine. The evaluation has been performed without any optimizations of the Tasklet system. We discuss these optimizations and extensions in the following two chapters.

# 6. Context-Aware Scheduling in Distributed Computing Systems

So far, we have introduced the Tasklet system. We have discussed the foundations of computation offloading, the requirements for a distributed computation system, as well as the design and the implementation of the Tasklet middleware. We have shed light on the processes of Tasklet creation, distribution, and execution. We have presented how Tasklets can be integrated into applications and how we can measure the performance of the Tasklet execution. We have, however, not yet mentioned how exactly Tasklets are scheduled and which resource is selected to execute which Tasklet. Instead, we cover Tasklet scheduling in this chapter separately for two reasons. First, scheduling is a complex matter and needs to be considered from multiple points of view. A scheduling strategy answers the question *what* gets offloaded *when* and *where to*. Each of the three elements, *what*, *when*, and *where to*, requires special attention as each of them can have a strong impact on the performance of the distributed computation system. A complex task scheduled to a slow provider, a data intensive task transferred via a weak link, or a time-critical task scheduled too close to its deadline will all result in unwanted outcomes. At the same time, these three elements are closely coupled and can hardly be worked on independently from each other. Thus, they are all discussed in this chapter. Second, the scheduling strategies are independent from the underlying distributed computation system. Even though, in the following, we discuss scheduling on the example of the Tasklet system, all strategies could be transferred to other systems as well. Thus, we present the general idea of the strategies first before we show how they are applied to the Tasklet system.

In the remainder of this chapter, we present several scheduling strategies all of which focus on a different aspect of scheduling. These strategies do not compete with each other but could be applied at the same time.

In 6.1, we use the concept of Quality of Computation (QoC) to increase the performance of computation offloading. We apply QoC in an environment with heterogeneous and error-prone devices that might leave the system at any point in time. We show how QoC can help to reduce the variance in execution times in these unstable environments. Further, we show how QoC can help to reduce the scheduling overhead. In 6.2, we propose a fault-avoidant scheduling strategy. In contrast to reactive, fault-tolerant schedulers, this strategy proactively avoids task allocations to providers that are likely to fail during the execution of the task. In 6.3, we implement a decentralized scheduling approach. As distributed computation systems accommodate large numbers of participants a central scheduling has to handle a great number of requests. To avoid performance bottlenecks and a single point of failure, we present a hybrid approach with decentralized scheduling decisions. In 6.4, we present three further strategies including the handling of priorities, task deadlines, and continuous task offloading.

## 6.1. Quality of Computation

In an ideal world, resource providers are always available, do not leave the system, and never stop processing a task mid-execution. Further, providers do not vary in their performance but build a homogeneous pool of resources which simplifies the scheduling decision. However, reality is often different. While cluster and grid systems might almost fulfill these criteria, edge computing environments, where end users provide their resources, do neither guarantee reliability nor are they homogeneous. Rather, they are typically highly heterogeneous in their performance and might leave the system at any point in time. This has tremendous consequences for distributed applications. Any application that depends on every single result requires that the execution of the task is eventually successful. Likewise, an application with time-critical tasks cannot accept delayed task executions. Thus, without further measures, unreliable and heterogeneous environments would remain unusable for these kinds of applications.

To overcome this situation, we use the concept of Quality of Computation (QoC) [157][1]. Developers can use QoC to define nonfunctional requirements for their applications. They can annotate tasks with QoCs and the scheduler takes these

[1] [157] is joint work with D. Schäfer, S. VanSyckel, J. M. Paluska, and C. Becker

annotations into account when making the scheduling decision. As discussed in 5.1.5, developers do not directly decide how the scheduler should enfore these goals but rather leave this responsibility to the scheduler itself. The goals only define *what* should be achieved but not *how* it should be achieved. Thus, as an example, they would state the QoC goal *'Reliability - guaranteed'* to request a guaranteed task execution. This is an important design decision as developers might not know in which environments their applications are executed. Schedulers, however, do know the environment and can adapt the strategies accordingly to enforce the QoC goals. Therefore, the schedulers choose from a set of QoC mechanisms. These mechanisms directly modify the behavior of the distributed computation system. In the example from above, the QoC mechanism *'Retransmission'* ensures that a task will eventually be executed by monitoring the execution state and by re-initiating the execution when the previous attempt failed. In this section, we discuss and evaluate the enforcement of two QoC goals, reliability and speed.

### 6.1.1. QoC Enforcement

We show how developers can use QoC to allow their applications to run in unreliable and heterogeneous environments. In the first scenario, an application depends on a reliable execution of tasks. As resource providers might leave the system mid-execution or stop the processing arbitrarily, the developer sets the QoC goal *'Reliability'*. There are two levels of reliability to choose from: *likely* which increases the probability that the tasks get executed, and *guaranteed*, which ensures that the task will be executed eventually. To enforce the *likely* level, the scheduler has multiple options. First, it could use the mechanism *'Instance Selection'* to pick a resource that is unlikely to fail. This is a complex approach which we discuss in Section 6.2. Second, it could use *'Multiple Execution'* where the same task is sent to multiple providers which increases the probability that at least one execution is successful [165]. However, neither of the approaches can guarantee success. One option for the scheduler to accomplish this is to activate *'Retransmission'*. This mechanism works on the level of resource consumers and providers. During task execution, providers keep sending heartbeats to consumers. When the consumer does not receive a heartbeat for a certain amount of time, it considers the provider broken and re-initiates the task execution on another

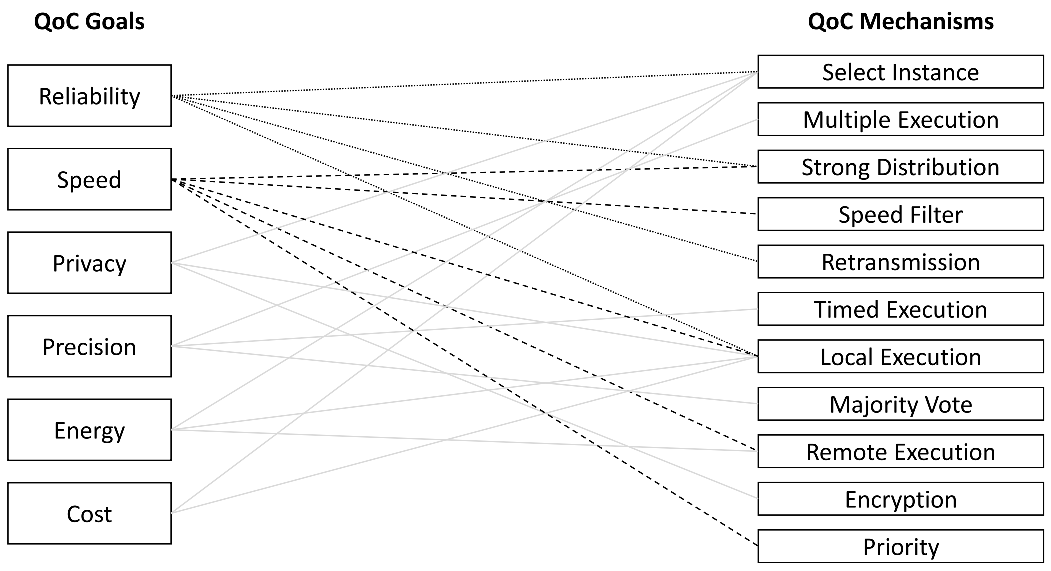


Figure 6.1.: QoC enforcement. The lines indicate which mechanisms the scheduler can choose to enforce a QoC goal. Therefore, it can select a single mechanism or a combination of multiple ones. The dotted lines show how *'Reliability'* can be enforced, the dashed lines indicate the same for the goal *'Speed'*.

provider. In case there is no network is available, a local execution can serve as a fallback option.

In the second scenario, the application requires the results in a timely manner. Delayed results are useless for the application and will be discarded. In a heterogeneous environment, a timely execution cannot be guaranteed. Depending on the processing speed of the provider, the execution time might vary significantly. It is important to mention that even with the QoC goal *'Speed'* the scheduler cannot guarantee that an execution meets a certain deadline. However, it has several measures to reduce the variance and to increase the mean execution time of tasks. Depending on the complexity of the task and the amount of data that has to be transferred for a remote execution, the scheduler can decide to execute the task locally or on a remote device. This decision highly depends on the context of the computing device such as its network connectivity. Therefore, it would not be reasonable to leave this decision to the application programmer. Instead, the scheduler takes this decision depending on the current context of the device. Another measure to reduce the execution time in heterogeneous environments is to start multiple executions of the same task at the same time and to consider the first result. This increases the chances to execute the task on a powerful device

| Device Type | CPU | Amount |
|---|---|---|
| Amazon EC2 - t2.mirco | 1 x 2.9 GHz | 100 |
| Amazon EC2 - m1.xlarge | 4 x 2.0 GHz | 10 |
| Amazon EC2 - c4.xlarge | 4 x 2.9 GHz | 40 |
| Amazon EC2 - c4.8xlarge | 36 x 2.9 GHz | 5 |
| Office Computer (Intel Q6600) | 4 x 2.4 GHz | 4 |
| Office Computer (Intel i7-4770) | 8 x 3.4 GHz | 1 |
| Intel NUC (Intel i3-4010U) | 2 x 1.7 GHz | 1 |
| Nexus 5/7 | SD S800/S4 | 2 |
| Total | | 163 |

Table 6.1.: Overview of the resource provider pool. The resources were geographically distributed across Dublin (Ireland), Frankfurt (Germany), and Mannheim (Germany).

and at the same time reduces the risk to be slowed down by selecting a single resource-poor provider. In case the performance of the providers is known to the scheduler, it can also use the *'Speed Filter'* to select only fast providers. Whether a provider is considered to be fast, depends on the composition of the current computing environment. As the performance of a provider fluctuates with its load, the filter only provides an estimate.

In the third scenario, the providers are not only heterogeneous but also unreliable. This does not change anything for the *'Reliability'* goal but has implications for those applications that require a timely execution. Each failure requires a retransmission and the execution has to start anew which adds to the total time [166]. We show how the scheduler needs to adapt its strategies to provide timely executions for the *'Speed'* goal in unreliable networks. Figure 6.1 shows which mechanism - or combination of such - the scheduler can select to enforce a QoC goal.

### 6.1.2. Evaluation Setup

We evaluated the different strategies in a real-world testbed consisting of a pool of heterogeneous devices. The pool included different Amazon EC2[2] instances, local office computers, Nexus 7 tablets, Nexus 5 smartphones, and an Intel NUC. From this pool, about 163 machines with a total of 518 CPU cores served as resource providers (cf. Table 6.1). One additional office PC acted as resource consumer only. For resource management, we applied an IBM Blade Center (Xeon E5345

[2]Amazon EC2: https://aws.amazon.com/de/ec2/, accessed: 20/03/2019

@ 2.33GHz) as the central broker. As in the evaluation of the Tasklet system, we used the Mandelbrot Set (MBS) application to test our strategies. A task in the MBS application is to render an image of 640 x 480 pixels based on a a recursive formula. To make results comparable we rendered the same part of the MBS throughout the entire evaluation. Each task in the application can be split up into several subtasks. We used two split factors, 20 and 40, and created one Tasklet for each of these subtasks. These split factors are chosen in a way that the execution of the overall task takes about 10 to 60 seconds. To simulated failures in the otherwise stable evaluation environment, we introduced an error rate that is either 25% or 50%. This means that the processing of each fourth, respectively each second, Tasklet got terminated mid-execution.

### 6.1.3. Scenario 1: Reliability in Erroneous Environments

In this first scenario, we requested reliability in an erroneous environment. Therefore, we selected the *'Reliability'* QoC goal with the parameter *'likely'* which means that the probability of a successful execution should be increased. It does, however, give no guarantees that the task will be executed eventually. Rather, each Tasklet is still executed in a best-effort manner and might not lead to a result. We used the *'Strong Distribution'* QoC mechanism and sent multiple copies of each Tasklet to different resource providers. Thereby, we increased the chance that at least one copy of each Tasklet got executed successfully. The fact that different resource providers were selected for each copy is important as otherwise two or more copies would have been lost if one provider failed.

The success ratio ($p_{success}$) can be determined mathematically. A task gets successfully executed when at least one copy of each subtask gets executed without a failure. With $\epsilon$ being the error rate of the providers, $s$ being the number of subtasks, and $c$ being the number of copies for each subtask $p_{success}$ can be computed as follows:

$$p_{success} = (1 - (\epsilon)^c)^s \tag{6.1}$$

In the formula, the term $1 - (\epsilon)^c$ computes the probability that at least one copy of a single split will be executed. The exponentiation with $s$ computes the probability that this is true for all splits and, thus, the whole task gets executed successfully.

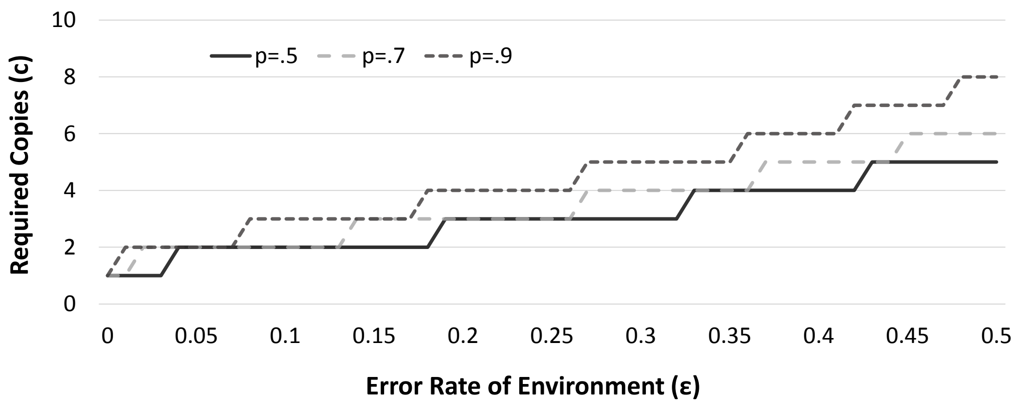


Figure 6.2.: Required number of copies for each of 20 subtasks to execute the complete task with a probability of at least $p$. For each copy one Tasklet is created.

The scheduler can use this formula to compute how many copies of a subtask are needed in order to achieve a success with at least $p$ percent. We isolate the number of copies $c$ by rearranging equation 6.1. Thus, $c$ can be computed as follows:

$$c = \lceil \log_{\epsilon} (1 - \sqrt[s]{p}) \rceil \tag{6.2}$$

We apply Equation 6.2 to learn how many copies are required to achieve 50, 70, and 90 percent probability in reliable environments (error rate 0%) and highly unreliable environments (error rate 50%). The results in Figure 6.2 suggest that for environments with an error rate of up to 25% four copies of each subtask are sufficient to reach a probability of at least 90%

We also performed an empirical evaluation of the *'Reliability'* goal with Tasklets in our real-world testbed. Therefore, we executed tasks in an environment with error rates of 25% and 50%. We varied the number of copies between 2, 4, and 8. Tasks are split into 20 and 40 subtasks. We ran each combination of the number of copies, split factor, and error rate which resulted in 12 settings. We measured the success rate of the executed tasks where at least one copy of each Tasklet has been completed. Figure 6.3 shows the results of 50 runs per setting. In general, the success rate increases with a high number of copies and a low number of subtasks. In environments with a 25% error rate, two copies are neither sufficient for 20 nor for 40 subtasks as only 18% respectively 6% of all tasks are executed successfully. The success rate decreases with the number of splits, as the probability that at

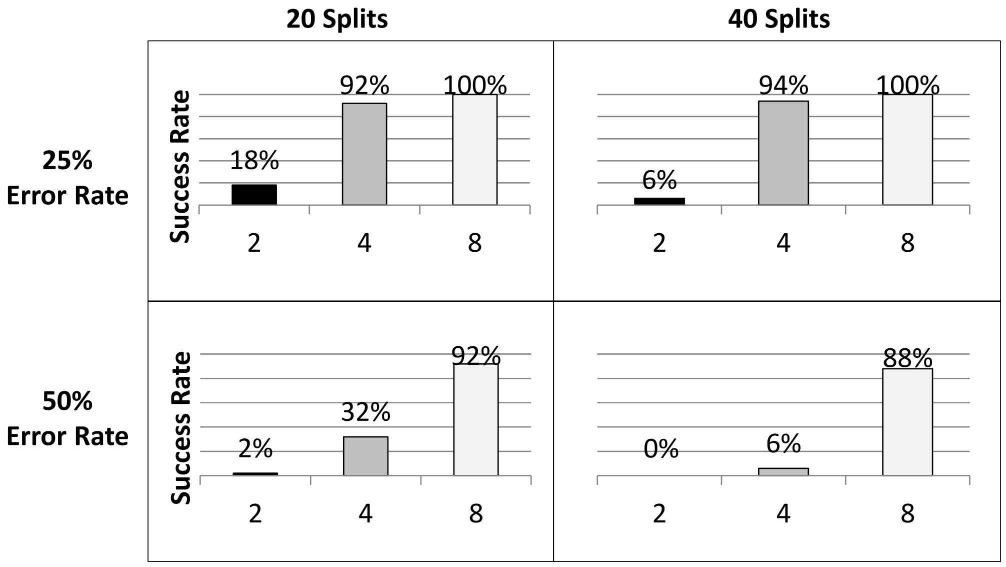


Figure 6.3.: Success rates for MBS computations in erroneous environments. Sending multiple copies of each Tasklet increases the probability that at least one copy of each Tasklet gets executed successfully. Further, splitting up the computation in multiple Tasklets reduces the success rate.

least one split breaks increases. The results suggest that 4 copies of each Tasklet are sufficient to achieve more than 90% chance that the task gets executed. In highly erroneous environments, however, 8 copies were required to achieve this probability. The results from the empirical evaluation are in accordance with the analytical study

### 6.1.4. Scenario 2: Speed in Reliable Environments

In the previous scenario, we only focused on the probability of a successful task execution in an erroneous environment. We did not take the speed of the task execution into account. In this scenario, the providers offer a reliable service but the application requests a timely response. Thus, the application developer selects the *'Speed'* QoC goal and splits up each task into multiple subtasks. As the environment is heterogeneous, some subtasks might be executed on fast providers whereas others run on resource poor devices. A task is completed when the last subtask is executed and all results can be aggregated. Thus, the slowest subtask is the bottleneck of the execution.

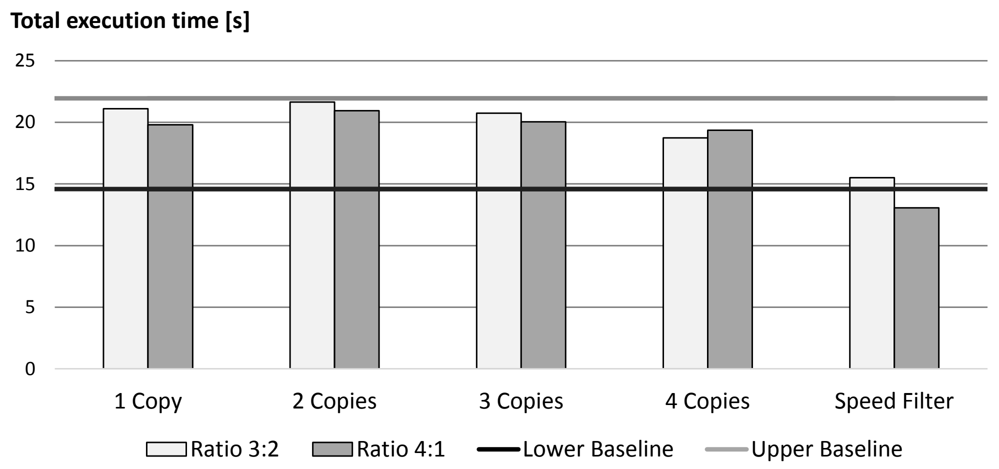


Figure 6.4.: Execution time of an MBS computation in error-free environments. The horizontal lines represent baselines in a slow (upper bound) and fast (lower bound) homogeneous environment. The bars indicate the result of an execution in a heterogeneous environment with different ratios of fast and slow machines. Increasing the number of copies leads to only a small benefit (cf. 1 vs. 2 to 4). Filtering for fast machines reduces the execution time substantially (compare *Speed Filter*).

We introduce two mechanisms to avoid these bottlenecks. First, we execute multiple copies of the same subtask on different providers and use the first result for each subtask. Thus, the chances increase that at least one copy of the subtask runs on a fast provider. Second, providers can be filtered by their speed. By selecting only the fastest available resources, we minimize the variance in the execution time of the subtasks.

To evaluate these strategies, we implemented them into the Tasklet system and ran the MBS application in our testbed. The testbed consisted of two different kinds of Amazon EC2 instances. As slow providers we used m1.xlarge instances with a clock rate of 2.0GHz. Fast providers were represented by c4.xlarge instances with a clock rate 2.9GHz. We split up each MBS task into 20 subtasks and created one Tasklet for each subtask. We computed baselines for the two types of instances. Therefore, we set up an environment with only slow providers to get an upper baseline for the execution time and another environment with only fast instances to retrieve a lower baseline. Both baselines can be found as horizontal lines in Figure 6.4. We further set up two different heterogeneous environments. In the first environment, we used 25 devices where the ratio between fast and slow instances was 3 : 2. In the second environment, we used 50 devices with a ratio of 4 : 1. First, we ran the MBS application with only one copy per subtask. The results in Figure 6.4 (1 Copy) show that in both environments, the average time for a task execution is close to the upper baseline, which represents the execution time in the slow environment. This is due to the fact that the execution of at least one subtask on a slow machine delays the overall execution. With more copies of the same subtask, the probability increases that at least one copy is executed on a fast machine. However, the results of the task execution with multiple copies (Figure 6.4, 2-4 Copies) do not show a significant improvement. As a second strategy to enforce the *'Speed'* QoC goal, we used the *'SpeedFilter'*. Only one copy per subtask was sent. This strategy shows the greatest benefit and reduces the execution time by 26%, respectively 34% compared to the execution without the speed filter.

Even though the findings of the quantitative evaluation indicate that the speed filter can help to reduce the execution time, the results in Figure 6.4 raise two questions that we will discuss in the following.

**A) Why do multiple copies do not reduce the execution time?** We would have expected that more copies of the same subtask increase the probability that at least one copy for each subtask is executed on a fast provider. As a result, the average execution time should be closer to the lower baseline. To analyze the behavior, we have simulated the scheduling in both environments. The results of the simulation suggest that with 4 copies instead of only one the probability increases from 0.0037% to about 59%. In the environment with four times more fast providers than slow ones, this probability even goes up to 96%. To understand why there are almost no improvements in the result, we performed a detailed analysis of the evaluation logs. We found out that the performance of the Amazon EC2 c4.xlarge instances heavily depends on the load of a single instance. As more copies lead to a higher load in the system, each instance has to execute more Tasklets. Whereas our office computers have proven to keep up the execution speed at a higher load, the EC2 instances slow down when more than one Tasklet is executed at the same time. Thus, the total execution time of a task does not increase even though more of the powerful instances have been selected.

**B) How can the speed filter outperform the results of the lower baseline?** As the lower baseline had been measured in an environment with 10 fast c4.xlarge instances the speed filter was expected to match these execution times at best. However, Figure 6.4 shows that applying the speed filter leads to even faster task executions. The answer to this question is closely related to the previous one. As the speed filter was applied in an environment with up to 40 fast instances, each instance had a smaller number of Tasklets to execute at the same time. Thus, the performance of these instances had not been impaired. In both environments, the same device types were used. However, for the speed filter, the workload had been distributed among more instances.

### 6.1.5. Scenario 3: Speed in Erroneous, Heterogeneous Environments

In the third scenario, applications are executed in an unstable environment where devices spontaneously leave the system. Devices were also heterogeneous in terms of their execution speed. As the applications require a timely execution, the *'Speed'* QoC goal is selected. Further, the application depends on the execution of every single task. The *'Reliability'* QoC goal with the option *'guaranteed'* does not only increase the probability that a task gets executed but ensures that the task is executed eventually as long as remote or local resources are available.

We used multiple QoC mechanisms to enforce both speed and reliability. A guaranteed execution can be achieved in two ways. First, a backup computation can be performed locally. In case the remote execution fails or the network connection gets lost, the results from the local backup can be used. However, we decided that executing all tasks locally might require more processing power than locally available and quickly drain the battery of the local device. As this solution contradicts the idea of computation offloading it only serves as a fallback option when no other solution is feasible. Thus, we do not proactively start the local execution. Second, task executions can be monitored and, in case of a failure, restarted on a different device. We decided to use this reactive mechanism as this reduces the load on the local device.

In terms of the execution time, this approach is not optimal. The detection of failures and the retransmission both cause delays. Further, the execution starts anew after each retransmission and the progress is lost. Thus, to provide a timely execution, we also applied the *Strong Distribution* mechanism and sent out multiple copies of the same subtask to increase the chance that at least one of these copies gets executed without the need for a restart. Finally, we also applied

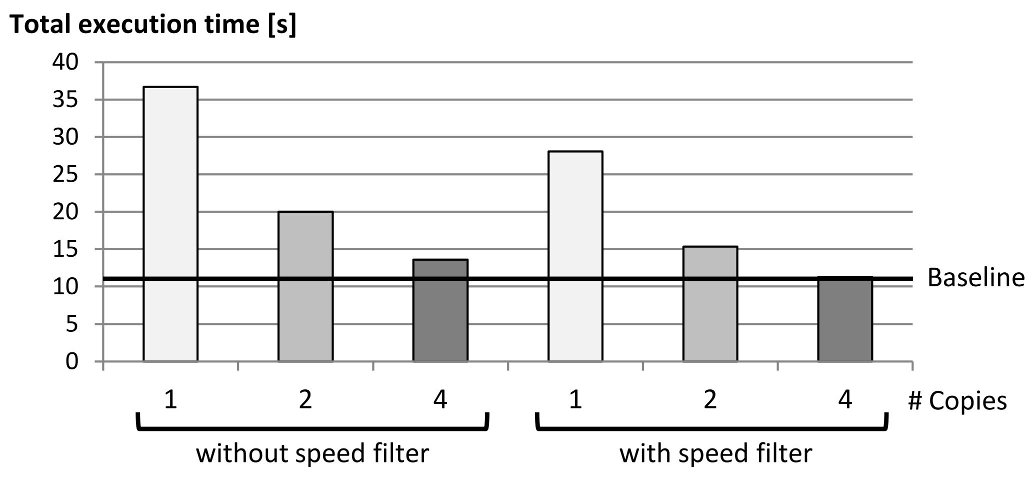


Figure 6.5.: Execution time of an MBS computation in erroneous environments. The baseline is measured in a stable environment. In an erroneous environment, sending multiple copies of a Tasklet results in a significant speedup (cf. 1 vs. 2 and 4). In combination with the speed filter, the execution speed hits the baseline even in highly unstable environments.

the *Speed Filter* to select only the most powerful devices within the resource pool as slow providers would be a bottleneck in the execution.

For this evaluation we ran the MBS application in an environment of 55 devices with an error rate of 50%. The clock rate of the devices varied between $2.0GHz$ and $2.9GHz$. We split up each task into 40 subtasks and sent out 1, 2, and 4 copies for each subtasks. Figure 6.5 shows the results of the 50 trials per setting. The horizontal line shows the baseline (11.07 seconds) that has been measured in a stable environment with no execution failures. Without *'Strong Distribution'* and *'Speed Filter'* the execution of the task takes more than 3 times in the unstable environment (36.67 seconds). By sending out multiple copies, this number can be reduced to 20 seconds, respectively 13.59 seconds. Activating the *'Speed Filter'* the execution time could be further improved. In the best case, with 4 copies of each subtask, the execution is only 2.07% slower than the baseline.

The results suggest that QoC can help to run time-critical applications in unstable environments. Without these execution guarantees these environments have been unsuitable as failures led to delayed results. The QoC mechanisms manage to eliminate the effect of these failures even in highly erroneous environments with error rates with up to 50%. However, the guarantees come at the cost of additional computations. Redundant executions and retransmissions lead to about 3 times more scheduled Tasklets for 2 copies and up to a factor of 5.5 when 4 copies were sent out.

## 6.2. Fault-Avoidant Task Allocation

In edge computing systems, where end-user devices might serve as computing resources, there are no guarantees about performance or reliability of these devices [39]. In contrast to traditional grid and cloud architectures [167, 168], edge computing systems typically do not define Service Level Agreements (SLAs) which serve as contracts between resource consumers and resource providers. Instead, resources might leave the system at any time and drop the current execution of a task. In the previous section, we have used the concept of Quality of Computation (QoC) to provide a timely and reliable task execution in such erroneous environments. To avoid delays caused by retransmissions, we have introduced redundant computations. Similar approaches of fault-tolerant scheduling focus on fault-detection strategies and recovery mechanisms [169] or replication [165]. While these approaches have proven to be effective, it does not come for free. The additional computations require resources and might congest the distributed computing environment. Hence, a more lightweight approach is preferable. The idea of fault-avoidant scheduling strategies is to anticipate resource failures and to proactively avoid scheduling tasks to those resources which might not finish the execution of a task successfully. In this section, we integrate a fault-avoidant scheduler into our Tasklet system [162][3].

### 6.2.1. Related Work

Since the advent of distributed computing, a lot of research has been conducted in the field of fault-tolerant scheduling. The approaches can be categorized into fault-aware and fault-avoidant strategies. In [170], Avizienis *et al.* provide a taxonomy of fault-tolerance and fault forecasting. A survey on fault-tolerance mechanisms in grid computing can be found in [171]. [165] uses replication of tasks to ensure a certain fault-tolerance level in mobile grid environments. They do not guarantee a reliable execution but increase the probability of success. [169]

[3] [162] is joint work with D. Schäfer, C. Krupitzer, V. Raychoudhury, and C. Becker

implements fault recovery where failed nodes will either be replaced or repaired. To increase the performance of fault discovery, the system introduces constraints on the executed tasks. In [172], a fault-aware scheduling for desktop grid systems is presented. The authors implemented eight fault-aware policies to increase the scheduling performance. Tang *et al.* [173] introduce a reliability-driven scheduling architecture to deal with node failures in heterogeneous environments. Berg, Dürr, and Rothermel discuss the concept of preemptable code offloading for mobile applications [148, 174]. As link failures represent one of the major drawbacks for mobile code offloading architectures their approach allows to benefit from offloading despite the presence of link failures. To achieve this, cloud servers send snapshots to the mobile device which continues the computation locally in case of a link failure.

However, as fault-recovery introduces a delayed execution, proactive approaches that predict failures even before they occur can increase the performance of distributed computing systems [175]. Rood *et al.* [176] provide fault-avoidance by availability prediction and replication. Their prediction is based on historical data about job completions. Lee *et al.* [177] predict the availability of mobile devices based on users' mobility patterns. They use these patterns to classify devices into three categories of availability. Ren *et al.* [178] use a semi-Markov process to predict the availability of resources. They monitor the free CPU load and free memory to detect state changes. In the approach of Chakravorty *et al.* [179], the authors assume that failures can be predicted and a running task can be migrated before a fault occurs. Duan *et al.* [180] use clustering and learning algorithms on workflow and historic data of resources to predict faults. Kang *et al.* [181] predict failures based on observed inter-arrival times of failures in managed machines. Sonnek *et al.* [182] and Damiani *et al.* [183] propose reputation-based scheduling approaches for peer-to-peer systems. Hummel *et al.* [184] implement proactive and reactive fault-tolerance. Proactive fault-tolerance is implemented by redundant executions. Reactive fault-tolerance is facilitated by resubmissions.

### 6.2.2. Failure Model

In distributed computing systems, there are multiple types of faults. Resource providers might terminate a task execution, leave the system, or return arbitrary results. As these failures happen randomly, there are no guarantees that tasks are executed successfully. In our failure model, we do not consider a malicious behavior of resource providers (Byzantine faults) but focus on those failures that are introduced by device fluctuation or by user-invoked terminations. Thus, our failure model takes the following situations into account:

**Explicit Leave:** Device owners might shut down the device at any time without paying attention whether a task is executed or not. This causes the execution to be terminated and the current progress to be lost. As the leave is intentional, there is enough time to inform the consumer about the termination of the task. This reduces the delay that is typically introduced by the detection of the failure.

**Implicit Leave:** Resource providers might leave the system without prior warning. Common examples are when the device crashes or loses its connection to the network. Latter is often the case for mobile devices. In these cases, the consumers have to detect the failure by appropriate mechanisms such as heartbeat channels.

**Task Termination:** Even though the provider does not leave the system, it might terminate the execution of the task during the execution. This can happen for two reasons. First, the device owner stops the execution manually. Second, the system withdraws resources from the execution since the resources are needed for local processes that have a higher priority. This behavior is common in scenarios where excess capacities are used.

### Fault-Awareness, Fault-Tolerance, and Fault-Avoidance

A system is *fault-aware* when it is able to recognize failures. Systems are not necessarily fault-aware as failures might not be noticeable by default. In the Internet Protocol (IP), packets are transferred in a best-effort fashion and packet losses might occur. To identify faults, routers send Internet Control Message Protocol (ICMP) messages to the sender when they have to drop an IP packet. When, in turn, the ICMP packet gets lost, the fault remains undetected given no further mechanisms are applied. Common measures to implement fault-awareness are heartbeat channels, acknowledgments, or error messages. Fault-aware systems might or might not take further actions when they detect a failure. *Fault tolerance* commonly is referred to as "*...the ability of a system to perform its function correctly even in the presence of faults.*" [171, p.88]. The system has to react to these failures to restore the functionality, for example, by restarting a process. Fault-avoidant systems attempt to anticipate failures and proactively take countermeasures to avoid their occurrence. Therefore, the system has to learn about the behavior of system components and to predict their behavior in the future.

### 6.2.3. Relevant Context Dimension

We integrate fault-avoidance in the scheduler of a distributed computation system. The scheduler assigns tasks from resource consumers to resource providers. As providers might fail, the scheduler needs to predict their behavior and assign tasks only to those providers which have a high probability to execute the task successfully. The behavior of providers might show a large variance ranging from highly reliable ones to others that have a very low chance of success. Thus, the scheduler needs to monitor the provider on an individual level instead of aggregating the data of all monitored providers.

The behavior of a provider is part of its context and a scheduler that considers this behavior is referred to as *context-aware*. In a first step, we determine which context dimensions need to be taken into account to predict the probability that a provider will execute a task successfully. Therefore, we consider the failures that we have defined in the failure model. First, task execution might fail because providers leave the system with or without prior notice. Second, providers might terminate the execution of tasks because resources are required for local processes. In the following, we focus on: (i) the residence time of providers in the system (stability) and (ii) the ability of providers to successfully execute Tasklets (reliability).

**Stability ($\mu, \sigma^2$):** Devices in the distributed computing environment vary in terms of the time they remain connected. While stable cloud resources are available most of the time, user-owned edge devices join and leave the system frequently. We assume that the residence time, or *stability*, for each device is normally distributed, where each provider has a certain mean ($\mu$) and variance ($\sigma^2$). A provider that shows a high variance is less predictable than a provider with a low variance. Stable providers show a high value for $\mu$ and a small value for $\sigma^2$. Figure 6.6 shows an exemplified distribution of the stability for two providers.

**Reliability ($\lambda$):** Besides leaving the system, providers can also cancel the execution of tasks while remaining connected. A provider that drops tasks frequently is

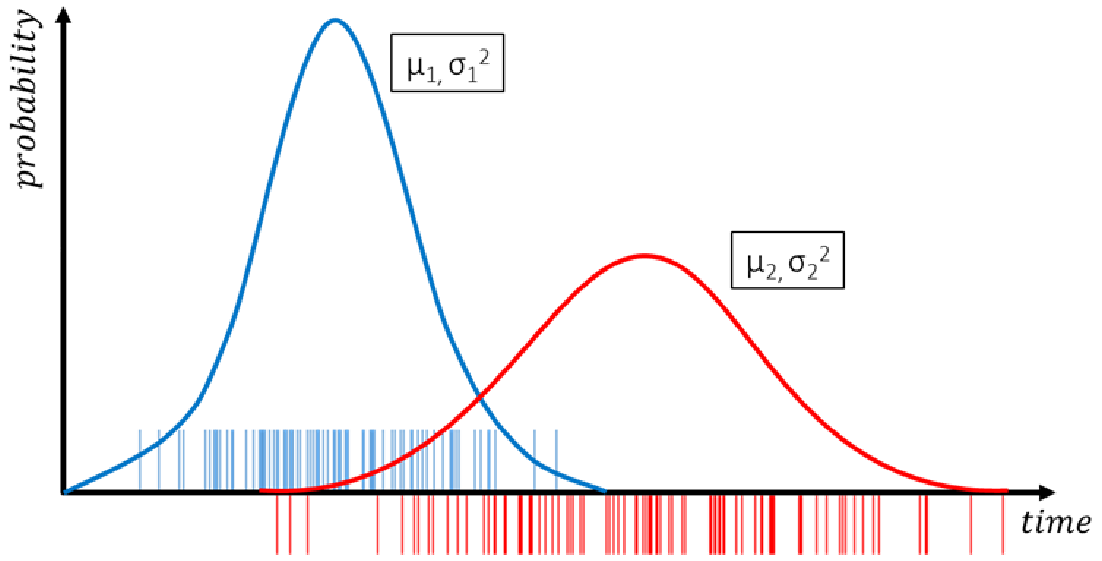


Figure 6.6.: Mean and Variance for two exemplary providers. The values are estimated by the monitoring functionality of the scheduler using historical information on connection times. The vertical bars show the measurement points for the respective provider.

considered less reliable than a provider that executes all tasks successfully. Within a distributed computing system, we define *reliability* as the performance of a provider to successfully execute tasks. For each successfully executed task, the reliability value $\lambda$ of the provider is increased. When the execution of a task is dropped, the reliability is reduced.

### 6.2.4. Design of a Fault-Avoidant Task Scheduler

We present a context-aware task scheduler that uses context information for a fault-avoidant task allocation. The scheduler can be integrated into any resource broker that has a global view on the entities participating in the system. It measures the context of the providers and makes adaptive scheduling decisions. For each task, a consumer sends a resource request to the broker. The broker performs the scheduling and selects the most suitable provider for the task. In accordance with other approaches in the domain of (self-)adaptive software, the adaptive scheduler integrates a feedback structure [185]. As feedback loop structure we use the MAPE loop [84]. This loop integrates functionalities for (i) monitoring the environment, (ii) analyzing the monitored data for the need of adaptation, (iii) planning the adaptation, and (iv) executing the adaptation [186].

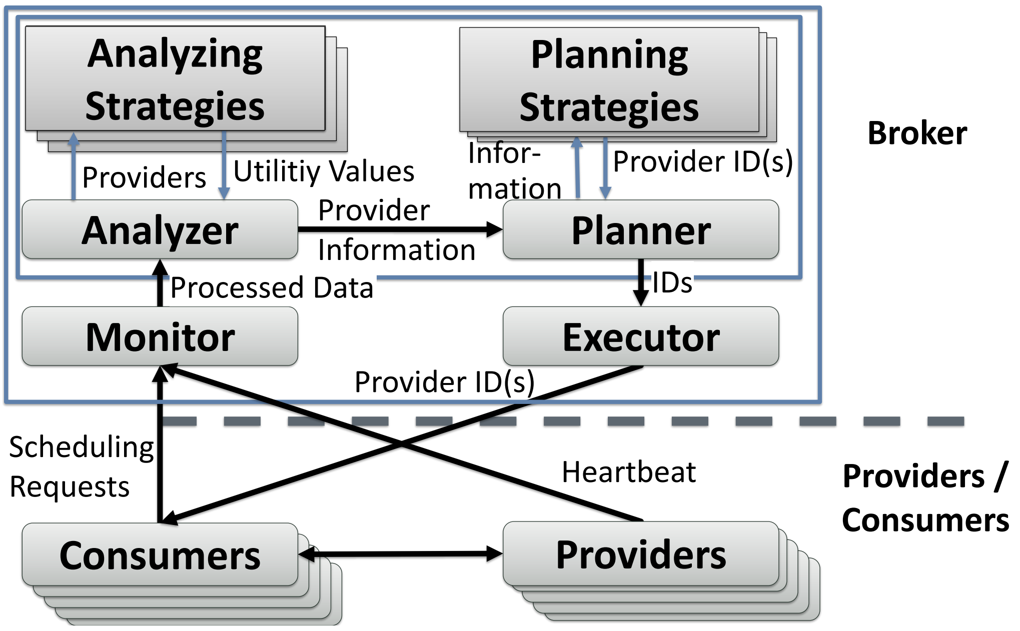


Figure 6.7.: System model of the scheduler exemplified with the Tasklet system. Further components of the broker are omitted.

Figure 6.7 presents the system model of the scheduler. The scheduler runs inside a resource broker that performs the resource management and serves as central scheduling entity within the system. Consumers and providers are connected to the broker which has a global view on all entities inside the system. The scheduler is implemented as a MAPE cycle extended by two modules that allow to select an analyzing or planning strategy at runtime. By following a modular approach, developers can easily exchange application-specific parts of the scheduler such as the collection and pre-processing of context data.

The monitor in the scheduler retrieves context information from the providers and consumers. It pre-processes the data and forwards them to the analyzer which computes a utility value for each provider. The utility value represents the suitability of a provider to execute a task. The analyzer can select a strategy from a set of analyzing strategies that might implement different utility functions. It forwards a list of rated providers to the planner which makes the final scheduling decision. Similar to the analyzer, the planner can choose from a set of planning strategies to optimize the decision. Following the Strategy Pattern [187], we encapsulate the algorithms for evaluating the current providers (analyzing) and decide which provider(s) should be used (planning). By using frameworks for self-adaptive system development for the implementation of the scheduler, the exchange of algorithms or parameters is possible at runtime [188]. The executor retrieves the scheduling decision from the planner and forwards the choice of providers to the consumer.

### 6.2.5. Monitoring

The monitor captures the relevant context information. When a provider enters the system, it starts sending periodic heartbeats to the broker and the monitor registers a new session for this provider. The session is over when no heartbeats are received within a certain interval. The monitor stores the duration of each session for each provider. It estimates the parameters of its normal distribution based on the observations $(x_{p,i})$, where $x_{p,i}$ is the i-th residence time of provider p in the system. $\mu_p$ and $\sigma_p^2$ are estimated as follows:

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

The monitor also needs to measure the reliability of the provider. Therefore, it compares the number of successfully executed tasks and cancelled executions. When providers leave the system before a task is executed they cannot inform the monitor about the failure. Hence, it is the responsibility of the consumer to give feedback about the success of each task execution. We implemented three approaches to calculate $\lambda_p$ for each provider p:

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

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

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

### 6.2.6. Analyzing

The analyzer retrieves a list of providers with estimations for $\mu_p$, $\sigma_p^2$, and $\lambda_p$ from the monitor and computes a utility value for each provider. Therefore, it selects a utility function that can be provided by the system administrator. Different utility functions consider the provider information in different ways, depending on which data is available and which context dimension is more relevant for the respective system. Here, we present three algorithms for calculating the utility of the providers: (i) reputation-aware analysis, (ii) dynamic reputation-aware analysis, and (iii) runtime-aware analysis. Further utility functions could be added.

**Reputation-Aware Utility Function:** We have developed a prediction algorithm that is based on the *reputation* of the providers. Azzedin et al. define reputation of a device as the '*[..] expectation of its behavior [...] within a specific context at a given time.*' [189]. In the context of the distributed computing system, the expected behavior of providers is to remain in the system and to execute tasks correctly. The prediction algorithm calculates a value that indicates the probability that a provider would leave the system or drop the task during execution. The algorithm has three input factors: the stability values ($\mu_p$ and $\sigma_p^2$) and the reliability value $\lambda_p$.

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

$$U_p = \frac{||\mu_p|| + (1 - ||\sigma_p^2||) + \lambda_p}{3} \tag{6.3}$$

**Dynamic Reputation-Aware Utility Function:** The analyzer takes the stability and the reliability of providers into account to improve the quality of scheduling decisions. However, this approach has some limitations. In this model, the three parameters to compute the utility all have the same weight. This might be a good choice for some environments but will not be optimal for all provider pools. Further, the dynamics of providers entering and leaving the system might change over time as well as their reliability. Depending on the composition of providers, the optimal scheduling strategy, and thus the calculation of the utility, might change over time. Thus, we extended the reputation-aware scheduling strategy by two features: First, we assigned weights $\alpha$, $\beta$, and $\gamma$ to the parameters of the utility function. Second, we made these weights adaptable, based on the current context of the system. In a system with high dynamism, the stability parameters $\mu_p$ and $\sigma_p^2$ are emphasized. In a system with many corrupt or unreliable providers, $\gamma$ is increased for increasing the significance of $\lambda_p$ at the cost of $\mu_p$ and $\sigma_p^2$. Thus, the adapted equation to compute the utility value looks as follows:

$$U_p = \alpha * ||\mu_p|| + \beta * (1 - ||\sigma_p^2||) + \gamma * \lambda_p \tag{6.4}$$

**Runtime-Aware Utility Function:** The reputation-aware analyzing strategies considered the stability and the reliability of the resource providers. This allows to select those providers that, in general, have the highest probability to execute a task successfully. However, the strategies do not take the time into account that providers have already been active. We refer to this as the current session time of a provider ($CST_p$). The probability that a provider remains in the system decreases with a high value of $CST_p$. Further, the complexity of a task affects the probability of a successful execution. Long-running tasks have a lower probability to be executed before the provider leaves the system. Hence, we take the complexity of the task into account. In general, the complexity of a task is not known but has to be estimated by either learning from historic data or by code analysis. Estimation methods for execution times have been discussed in [190].

In our model, we assume that the runtime of a task $R_t$ is estimated by the consumer and provided to the scheduler. With more knowledge about the runtime of a task, we can estimate how likely it is that a provider will remain long enough in the system. When a task is dropped during execution, the current progress of the execution is lost and the task has to be restarted. Tasks with a longer runtime suffer more from a drop than short ones. For short tasks the stability of providers

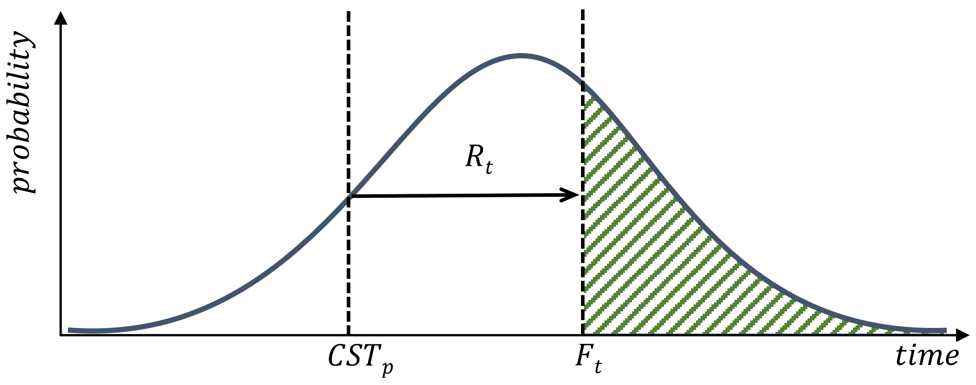


Figure 6.8.: Computation of the probability that a provider would remain in the system until the execution of the task has finished ($F_t$). This probability is used to compute the utility of the runtime-aware strategy. ($CST_p$ = current connection time of the provider, $R_t$ = runtime of the Tasklet)

is less critical. The runtime-aware strategy should prevent the failure of complex tasks and thus benefit the overall system performance.

We compute the utility of providers by a linear combination of two parameters: 1) the probability that the provider remains in the system until the end of the task execution and 2) the reliability of the provider. Whereas the reliability is computed as in the utility functions above, the probability of a successful execution depends on the estimated stability of providers and the runtime of the task. Using the current connection time ($CST_p$) and $R_t$, we predict the point of time $F_t$ when the execution of the task will be finished. Under the assumption that the connection time of providers is distributed normally, we can use $CST_p$ and the two parameters $\mu$ and $\sigma^2$ to calculate the probability that a provider is still active at $F_t$. Figure 6.8 illustrates this computation. The likelihood ($\tau_{p,t}$) that a provider remains long enough in the system to execute the Tasklet successfully can be computed as follows:

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

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

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

As a result, the overall utility $U$ can be computed as a linear combination of the likelihood $\tau$ and the reliability $\lambda$:

$$U_{p,t} = \epsilon * \tau_{p,t} + \gamma * \lambda_p \tag{6.5}$$

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

### 6.2.7. Planning

The planner makes the final scheduling decision and assigns a resource provider to each task. Therefore, it selects an appropriate planning strategy. We have implemented four different planning strategies: (i) naive, (ii) active, (iii) idle, and (iv) utility. Table 6.2 summarizes the differnt planning strategies and the context dimensions they take into account. Further planning algorithms can be added.

**Naive:** The naive planner randomly selects providers from the list of known providers. It does not take any further context information into account. This strategy serves as a baseline to identify the optimization potential of the context-aware strategies.

| | State | Workload | Stability | Reliability |
|---|---|---|---|---|
| Naive | | | | |
| Active | x | | | |
| Idle | x | x | | |
| Utility | x | x | x | x |

Table 6.2.: Planning strategies and context dimensions used

**Active:** The second planning strategy randomly selects active providers from the list provided by the analyzer. It excludes those providers that do not have an active session to avoid delays that are caused by unsuccessful scheduling attempts. This planning strategy requires knowledge about the current state of providers which is monitored by using heartbeats.

**Idle:** Providers that are considered in the scheduling decision might be busy running tasks or their resources are used locally. Thus, even though their state is active, they would not execute tasks for other consumers. Therefore, this planning strategy considers the current workload of active providers and selects idle planners only. Load information can be piggybacked on heartbeats from the providers.

**Utility:** The previous planning strategies do not consider the utility that was computed by the analyzer. The utility-based strategy selects the idle providers with the highest utility value to increase the probability that a task is executed successfully. As a result, providers with a high utility value are selected more frequently and the overall performance is expected to increase. The strategy can be combined with each analyzing strategy.

### 6.2.8. Evaluation

We ran the evaluation of the scheduling strategies in a simulated distributed environment that matched the structure of the Tasklet systems. The results of the evaluation can be applied to the Tasklet system but are not limited to it. In contrast to the evaluation of the Tasklet system that we ran in a real-world testbed with more than 150 physical devices and cloud instances in [157], we decided to use a simulated environment for the evaluation of the fault-avoidant scheduling strategies. This decision had two reasons. First, to measure the performance of the scheduler reliably, the system needs to span a large amount of resources. In real-world testbeds, it is extremely costly to set up hundreds or thousands of devices. Second, in the evaluation we simulate node failures, task drops, and several context states to demonstrate how the scheduler reacts to different system conditions. Controlling these parameters in a real-world testbed is hardly feasible.

The simulation model consisted of the three entities of a distributed computing system: providers, consumers, and brokers. For this evaluation we used a centralized pattern with a single broker that handled all resource requests from the consumers. There were 1000 providers that randomly entered and left the system randomly based on stability and dropped a task with the probability $\delta_p$. 100 consumers offloaded tasks with different runtimes to the brokers in random intervals. In each evaluation trial, we executed $50,000$ tasks. To measure the performance of the scheduling decisions, we counted the number of faults $n$ that happened per task. An $n$ of 0.5 means that every second a task had to be restarted

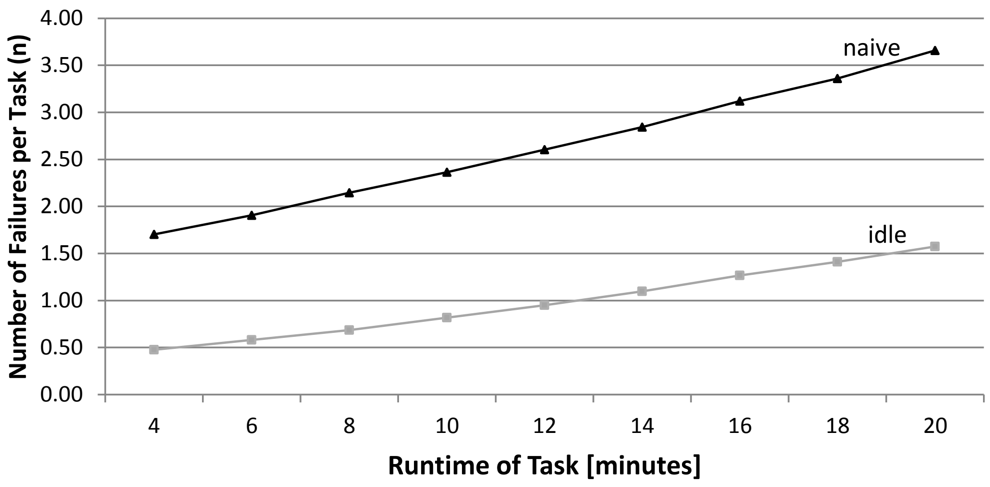


Figure 6.9.: Baseline measures for the simple scheduling mechanism in the distributed computing environment. The *naive* baseline has no context information at all. The *idle* strategy has knowledge about whether the provider is active and idle.

due to a fault. An $n$ of 2 means that each task had to be restarted twice. The goal was to minimize the number of faults as each task that needs to be restarted introduced a delay.

### Baseline

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

The *naive* baseline strategy represents a best-effort scheduling approach that does not make use of any context information at all. It randomly assigns a task to a provider from its resource pool and forwards this information to a consumer. This provider might have left the system or is currently busy executing other tasks. As a result, it might not execute the task and the consumer has to request a new provider. The *idle* baseline strategy uses the context information whether the provider is currently active and whether it is idle or not. While this approach requires some context-awareness it reduces the number of unsuccessful task requests from consumers to inactive or busy providers.

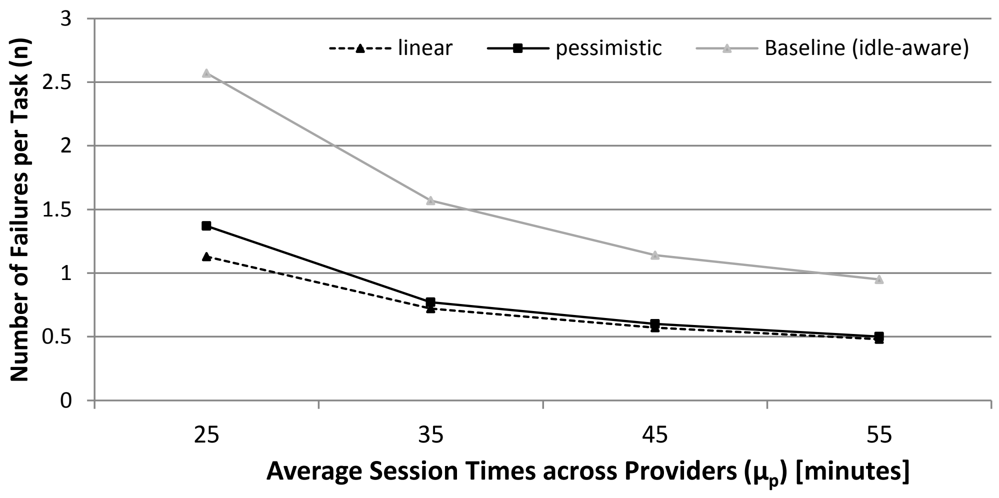


Figure 6.10.: Reputation-aware scheduling for different system environments (highly dynamic to less dynamic). Both strategies (linear and pessimistic) clearly outperform the baseline and further reduce the number of failures.

Figure 6.9 shows the results of the baseline evaluation. The plots indicate the average number of failures per task, sorted by the execution time of tasks. The results provide two insights into the system: First, implementing some context-awareness already increases the quality of scheduling decisions significantly. By monitoring the state of the providers, many faults, and thereby retransmissions, can be avoided. Second, tasks with a higher execution time are more likely to fail and have to be restarted more frequently. This behavior was expected as the probability that a provider leaves during task execution increases with the execution time.

### Reputation-Aware Strategy

The baseline evaluation has shown that context information can improve scheduling decisions. In the next step, we evaluated the reputation-aware strategy that selects the provider based on the utility that is computed according to Equation 6.3. To adapt the reliability parameter $\lambda$, we used the *linear* as well as the *pessimistic* approach. We simulated four different system environments, ranging from a highly dynamic one (with providers having a mean session time of 25 minutes) to a more stable environment with sessions lasting in average 55 minutes. The results in Figure 6.10 show that the reputation-aware strategy clearly outperforms the *idle* baseline which is shown as a reference. With a higher stability, the failure rate

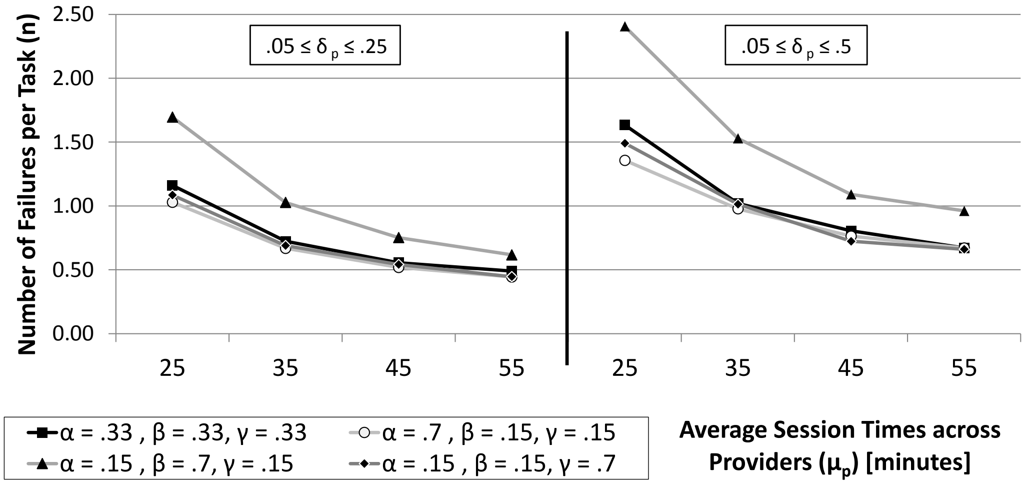


Figure 6.11.: System-aware scheduling for different environments. The plots show scheduling results from different combinations of weights in Equation 6.4. The tests have been performed with two different task drop rates $\delta_p$.

decreases and less tasks need to be restarted. Further, the *linear* adaption for the reliability ($\lambda_p$) performs better than the *pessimistic* strategy. In the following, we will continue with the *linear* strategy.

### Dynamic Reputation-Aware Strategy

As the system characteristics might change over time, the scheduler should be able to adapt to the new situation. In the dynamic reputation-aware strategy, the scheduler monitors the system context and adapts the weighting parameters $\alpha$, $\beta$, and $\gamma$ (see Equation 6.4), if necessary. We ran the dynamic reputation-aware scheduler in the same four system environments as in the previous test. In each environment, we tested different combinations of the weighting parameters. The results in Figure 6.11 (left) show that the algorithm performs best if the highest weight is put on the expected session time of the providers. This becomes even more relevant when the system is very dynamic (average session time = 25 minutes) and providers join and leave frequently. Putting too much weight on the variance, the performance of the scheduler decreases substantially. The dynamic reputation-aware strategy outperforms the baseline (not shown here) and slightly improves the result from the reputation-aware strategy.

The drop rate $\delta_p$ defines the ratio to which a provider drops a task during execution. So far, each provider got assigned a $\delta_p$ between 0.05 and 0.25. For the dynamic

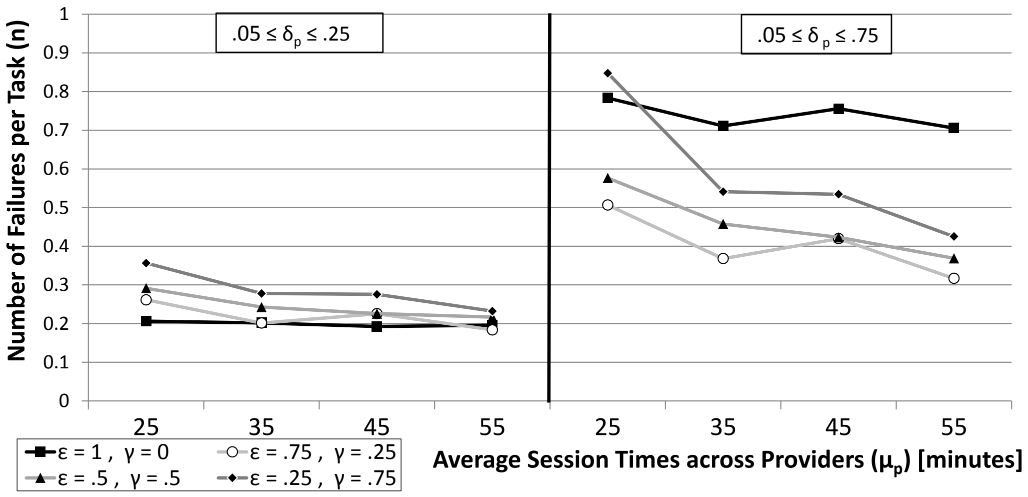


Figure 6.12.: Runtime-aware scheduling systems in four different environments. In an environment with low drop rates $\delta_p$, the strategy $\epsilon = 1, \gamma = 0$ is optimal. For less predictable environments ($.05 \leq \delta_p \leq .75$), the parameters have to be adjusted.

reputation-aware scheduling, we also tested a second scenario with a $\delta_p$ between 0.05 and 0.5. As a result, the average number of failures per Tasklet increased. The selection of the optimal weights remains the same (see Figure 6.11 (right)).

### Runtime-Aware Strategy

In the last step, we evaluated the runtime-aware strategy. It does not only take the context of providers and the system into account, but also uses the context of the consumer, as it considers the runtime of a task for the scheduling decision. Similar to the two approaches before, we tested the runtime-aware strategy in four environments with different average session times of the providers. We used different weights on the parameters $\epsilon$ and $\gamma$ in Equation 6.5. The results in Figure 6.12 (left) show that the best results can be achieved when the full weight of the function is put on $\epsilon$, and thus the scheduling decision only depends on the likelihood of a provider to remain in the system long enough. The performance of this mechanism remains stable even when the system becomes more dynamic and providers enter and leave the system more frequently. However, when we increase the drop rates $\delta_p$ to values between 0.05 and 0.75, this strategy does not perform well anymore (see Figure 6.12 (right)). Instead, in highly unreliable environments, it is beneficial to take the reliability measure $\lambda_p$ into account and put some weight on the factor $\gamma$. It is the responsibility of the scheduler to monitor the environment and to adapt the scheduling parameters accordingly.

### Discussion

In summary, the results from the three fault-avoidant scheduling strategies show that taking the context of consumers, providers, and the system into account, can substantially increase the performance of a distributed computing system. However, the performance increase does not come for free. Monitoring the context, analyzing the data, and selecting the optimal provider requires additional overhead that increases with the number of context dimensions. Thus, for each each system, the right strategy needs to be selected.

## 6.3. Decentralized Scheduling

In the previous chapters, we have performed the scheduling decision on centralized brokers. However, in distributed computing systems with a large number of heterogeneous nodes, task allocation becomes a highly complex problem. Schedulers have many options to assign a resource to a task and typically need to make this decision under time constraints as resource consumers expect a timely response. This is of even greater importance when the duration of tasks is in the order of seconds or sub-seconds [123]. In addition, the quality of the scheduling decisions has a great impact on the performance of the distributed computing system. Mismatches between tasks and resource providers can result in delayed or, even worse, cancelled executions. In environments where nodes dynamically enter and leave the system, outdated information might lead to scheduling failures as resource providers might not be available anymore. As a result, the scheduling procedure has to be repeated and another delay is introduced. In centralized environments, a single resource broker that has global knowledge about the entities in the system makes the scheduling decision.

Figure 6.13 shows the scheduling procedure in a hybrid peer-to-peer system where the scheduling is performed by a central broker. The central resource broker has global knowledge about the entities in the system and responds to resource requests from the consumer. Consumers and providers directly exchange tasks and results. As the broker has knowledge about all available providers, it can make optimal scheduling decisions. However, this approach suffers from the typical weaknesses of centralized systems as the scheduler represents a single point

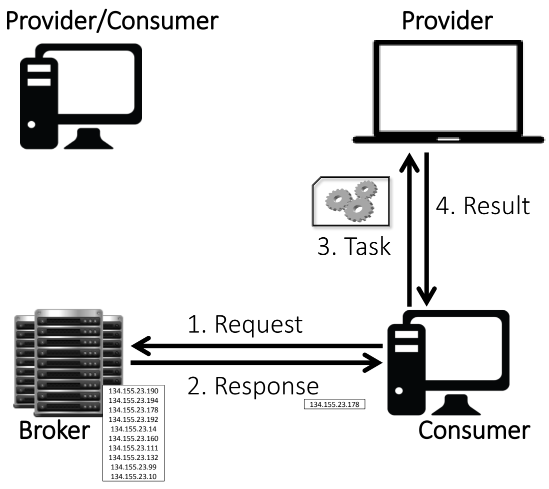


Figure 6.13.: Centralized scheduling in a distributed computing system. The central resource broker has global knowledge about the entities in the system and responds to resource requests from the consumer. Consumers and providers directly exchange tasks and results.

of failure and a performance bottleneck. Further, consumers have to request a resource for each task. The execution of the task cannot start until the client/server communication for the resource request has finished. We therefore propose a decentralized scheduling approach for the Tasklet system [191][4].

### 6.3.1. Related Work

In the literature, there is a vivid discussion of the advantages and shortcomings of a centralized task allocation. Whereas one side argues that current trends such as cloud computing make central architectures scalable and reliable [192], others suggest that only a distributed solution can provide both responsiveness and availability [123, 193, 194].

Dogar *et al.* introduce Baarat, a decentralized task-aware scheduler for data center networks [194]. Task-awareness means that the system knows the size of the task and the number of subtasks. It uses task serialization to avoid network bottlenecks and focuses on reducing the average as well as the tail task completion time. The scheduling decision is fully decentralized but requires a single entrance point for each type of tasks. In [195], Ogston *et al.* present a scheduler for massively scalable systems with 500 to 32,000 agents. Agents are resource-poor devices

[4] [191] is joint work with D. Schäfer, and C. Becker

that are required to cooperate with others to complete tasks. Similar agents form groups to cooperate. The algorithm works fully decentralized and agents only use knowledge from their direct neighbours. However, the groups themselves need to be managed by a central component. By limiting the size of the groups the scheduling results can be improved [196].

Other approaches build on top of distributed hash tables (DHT) such as Content Addressable Network (CAN) [197] and Chord [198]. DHTs provide a highly, robust, and scalable resource management [199]. Kim *et al.* build a so-called Rendezvous Node Tree based on a Chord ring to perform task allocation in fully decentralized heterogeneous computational environments [193]. Each parent node in this tree aggregates information about providers and responds to resource requests from its child nodes. Provider information includes the speed of the CPU and the amount of memory available. The authors showed that the matchmaking between tasks and providers is effective but cannot compete with the performance of a centralized scheduling solution. Jackson *et al.* present a completely decentralized algorithm based on CAN [200]. The algorithm manages cluster sizes where a cluster is a group of providers that can be used together for parallel processing of a task. The approach performs comparably to a centralized solution but targets at tightly-coupled parallel tasks only that require communication. Lee *et al.* use local task rescheduling and internode scheduling to maximize the throughput and perform load balancing across heterogeneous multi-core desktop grids [201]. Local task scheduling uses remaining available resources on one node and allows tasks to skip the queue in order to use resources more efficiently. Internode scheduling extends this mechanism to multiple nodes. The results indicate that a greedy central scheduler provides a better performance.

In [184], Hummel *et al.* implement a robust decentralized task scheduler for mobile peers in ad-hoc grids. The resource management is performed by a virtually shared memory. An autonomous local scheduler allows to make scheduling decisions in a decentralized manner. The performance of the task allocation is tightly coupled to the performance of the virtual shared memory architecture which is not further evaluated. Iamnitchi *et al.* introduce a decentralized scheduling strategy for dynamic grid environments [202]. The authors argue that in dynamic grid environments resource information of other peers soon becomes stale. Thus, they propose a strategy where each peer autonomously decides whether to execute a task or forward the execution request. The simulation results with more than 30,000 nodes suggest that learning about the performance of neighbour nodes increases the quality of scheduling decisions. Still, one resource request requires, on average, more than a dozen hops. Cardellini *et al.* extend the Storm realtime computation system [203] by a distributed QoS-aware scheduler [204]. QoS information about nodes includes the current utilization and availability. They are exchanged by a gossip-based dissemination protocol. The task placement is based on the idea of cost spaces that were introduced by Pietzuch *et al.* [205]. The distributed approach outperforms the centralized default scheduler in Storm.

In [123], Ousterhout *et al.* present Sparrow, a stateless, distributed low latency scheduler. The scheduler aims at systems with response times in the magnitude of sub-seconds. The scheduler samples multiple worker nodes and forwards a task to the one with the shortest queue. Further, it applies late binding, a form of work stealing where worker nodes place reservations for tasks. The results indicate that Sparrow presents a viable alternative to centralized schedulers. However, this only holds for low latency clusters since otherwise probing and replies would require too much time.

Mohaisen *et al.* introduce a hybrid task scheduling model including a centralized scheduler which has global knowledge [206, 207]. Before a task gets executed, the resource consumer sends a request to the centralized scheduler which replies with a list of available resource providers. Thus, for each request, a full round trip time between consumer and centralized scheduler is required.

### 6.3.2. Towards Decentralized Tasklet Scheduling

From related work, we have identified three major drawbacks of central schedulers. First, they are not scalable as they need to make a large number of complex decisions in short time and have to deal with countless messages. Second, they constitute a single point of failure as tasks can only be scheduled in one place. Third, they introduce latencies, as for each scheduling decision, multiple messages have to be exchanged.

We argue that for the Tasklet system the latter is the most severe drawback among the three. Scalability can be achieved by running the resource broker on adaptive cloud resources that scale with the workload. Further, when one broker gets overloaded, a second broker can be spawned to handle a part of the resource pool. Both pools then have a single central scheduler and the size of the pools remains manageable. Reliability is at risk when central brokers are single points of failure. Even though modern systems show an uptime of more than 99.99 percent there are situations left in which the whole system is unusable. Thus, we argue that even in the presence of central entity in the system, a certain level of fault-tolerance is required. The level of fault-tolerance has to ensure that the system remains usable for the downtime of the central entity. However, the most critical point is the latency that is introduced by the client/server communication when a consumer requests a resource from the broker. To remove this delay, the consumer needs to have some local knowledge about available resources. At the same time, the consumers should not be involved in the resource management as this would require extensive communication between all entities in the system.

We propose a system architecture that maintains a central entity for the resource management but that also facilitates decentralized scheduling decisions. We introduce three different techniques for a decentralized scheduling in the Tasklet system. These approaches are not mutual exclusive but can be combined to achieve the best scheduling results. The three techniques are cache lists, multi-hop scheduling, and multi-level scheduling. They will be discussed in the following.

### 6.3.3. Scheduling with Cache Lists

So far, consumers in the Tasklet system requested a resource from the central broker which performed the scheduling decision and responded to each resource request individually (see Figure 6.13). Thus, the delay before a task could get executed included at least one round trip time between the consumer and the broker. The central scheduler has the advantage that resource consumers and providers have a well-known registry in the system. Due to its prominent role, the central scheduler has global knowledge over all entities in the system and knows about the availability and performance of the computing resources. We propose a scheduling architecture that eliminates the need for a resource request to the broker. Instead, the broker asynchronously shares its knowledge about connected providers with the consumers in form of cached resource lists, or cache lists for short. The management of the resource, which includes the registration

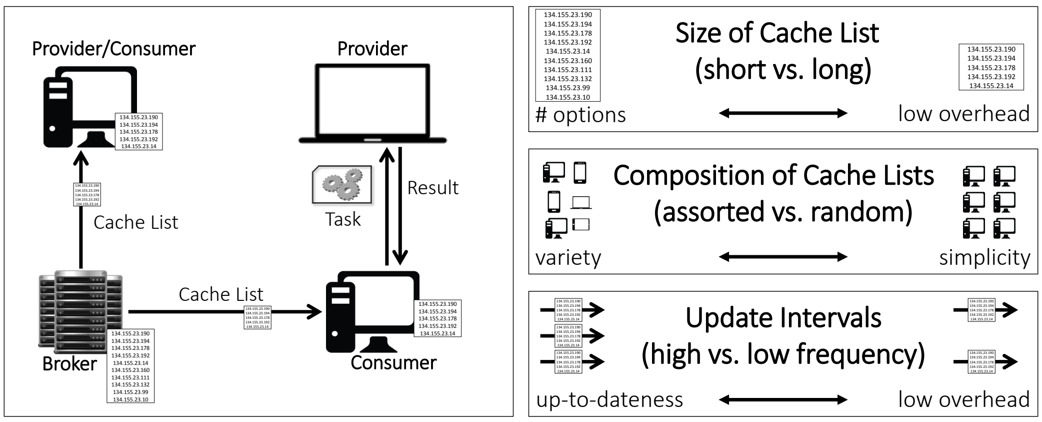


Figure 6.14.: Scheduling with cache lists. The broker periodically disseminates lists with provider information to all consumers. Instead of sending resource requests to the broker for each Tasklet, consumers cache these lists and locally select a suitable provider for the Tasklet execution. Trade-offs between performance and overhead are shown on the right.

and monitoring of resource providers is still facilitated by the central broker which maintains a global view of the resource pool.

The broker does not respond to each individual resource request from a consumer but rather periodically sends resource lists to the consumers. The consumers cache these resource lists and use them for the selection of suitable providers for Tasklet executions. Instead of sending a resource request to the broker, consumers look up a provider in the cache list and directly send a Tasklet execution request to the selected provider. Figure 6.14 (left) shows the decentralized scheduling with cache lists. While this approach avoids the per-Tasklet communication between consumers and brokers, it also carries some risks. First, the cached lists can easily become outdated as providers might leave the system at any time. Second, if the cache lists are too small, consumers might not find an appropriate provider. Third, the distribution of cache lists might introduce a significant overhead. As a consequence, several trade-offs between overhead and performance emerge (see Figure 6.14 (right)). In the following, we discuss the parameters of this architecture that must be adjusted carefully to guarantee a performant yet lightweight decentralized scheduling.

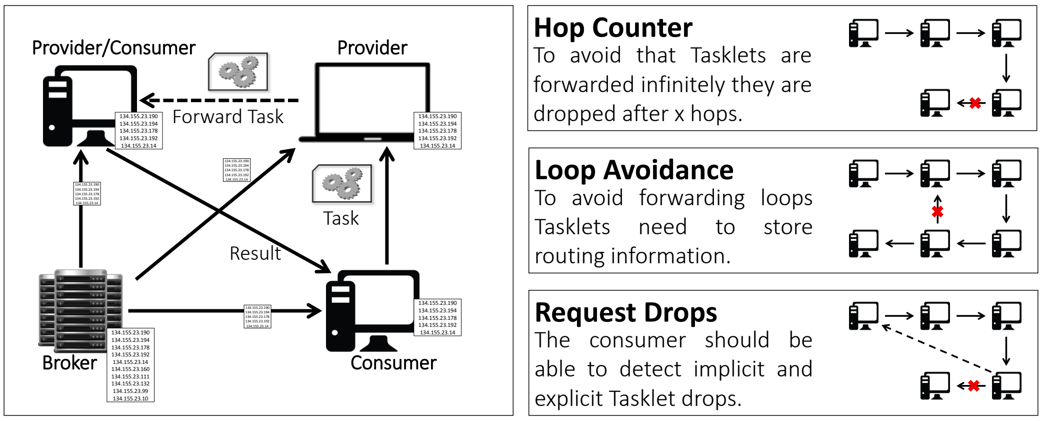
**Size of Cache Lists:** Brokers maintain a complete and up-to-date view on their resource pool. To allow for local scheduling decisions, they share this knowledge with the consumers by periodically distributing cache lists. This might result in a large amount of data transfer between the broker and the consumers. To reduce the communication overhead that is used for resource list propagation, consumers only retrieve a subset of this list. As each consumer gets a different share of the complete list, the load can be distributed equally across all providers. However, if the lists become too small, consumers might not be able to find suitable providers and have to send a resource request to the broker.

**Composition of Cache Lists:** Brokers might store multiple properties of each resource provider. Providers may vary in their availability, their hardware, and also in their connectivity. These properties can be used in a context-aware scheduling system. As the broker only forwards a share of the overall provider list to each consumer, the composition of the list might have an impact on the performance of the scheduling decisions. Each Tasklet might have several requirements for execution that only some of the providers fulfill. The composition of the cache lists can be managed in two ways. Providers can be either picked randomly or the mixture of providers can be balanced based on their properties. While the first approach reduces the complexity of creating provider lists, the second approach guarantees a fair propagation of providers to the consumers, which might be more likely to find a suitable provider for the execution of their Tasklets.

**Update Intervals:** Due to the fact that providers might leave the system at any time, the cached resource lists of the consumers eventually become outdated. Depending on the degree of dynamism, the speed of this process varies. As a result of dynamism, consumers are unable to reach the selected providers. To keep the cache lists up-to-date, brokers periodically send updates to the consumers. The interval of these updates represents another trade-off between up-to-dateness and overhead caused by the propagation. Instead of using fixed intervals, the brokers can monitor the degree of dynamism in the system and adapt the time interval between two updates accordingly. While this approach allows for a context-aware adaptation of the update intervals, the monitoring introduces further overhead.

### 6.3.4. Multi-Hop Scheduling

Scheduling with cache lists eliminates the delay that is caused by the resource request from the consumer to the broker. This round trip time can be saved if the selected provider is ready to execute the Tasklet. However, if this provider is



**Hop Counter**
To avoid that Tasklets are forwarded infinitely they are dropped after x hops.

**Loop Avoidance**
To avoid forwarding loops Tasklets need to store routing information.

**Request Drops**
The consumer should be able to detect implicit and explicit Tasklet drops.

Figure 6.15.: Multi-hop scheduling. The consumer sends a Tasklet for execution to a provider from its cache list. If the provider cannot execute the Tasklet, it forwards it to another provider from its own cache list. Challenges of this approach are shown on the right.

not able or willing to execute the Tasklet, it rejects the execution request and the consumer has to select another provider. This also results in a delay of at least one round trip between the consumer and the provider.

To avoid this delay, we introduce the concept of multi-hop scheduling as an extension to cache lists. In this approach, providers receive cache lists from the central broker as well. In case a provider does not execute a Tasklet, it does not send a negative feedback to the consumer, but forwards the Tasklet to another provider from its own cache list. This approach does not only save the second half of the round trip, but also reduces the workload for the consumer.

Figure 6.15 (left) shows the scheduling procedure. The broker sends cache lists to both consumers and providers. The consumer selects a provider for Tasklet execution. The provider is not ready for execution but forwards the Tasklet to another provider. Besides its benefits, multi-hop scheduling introduces further challenges that are illustrated in Figure 6.15 (right) and will be discussed next.

**Hop Counter:** If no suitable provider is available, the Tasklet would be forwarded between providers infinitely and would congest highly-utilized systems even more. Thus, we introduce a hop counter that gets decremented with every forwarding. When the counter reaches zero, the forwarding is interrupted and the provider which currently has the Tasklet informs the consumer. The consumer can then decide how to further proceed. The optimal initial value of the hop counter requires knowledge of the parameters of the underlying system such as the fluctuation of providers and communication latencies.

**Loops:** Tasklets requests might be forwarded between a small group of peers and loops might occur. This reduces the chance for a Tasklet to be successfully scheduled since the same providers are requested multiple times. To avoid loops in multi-hop scheduling, we extend the header of the execution request by a field of visited providers. These providers are then excluded in the forwarding process.

**Request Drops:** Execution requests can be dropped in two ways: 1) The hop counter runs out and 2) a provider crashes during the forwarding process. In both cases, the consumer needs to know about the failure. While in the first case, the respective provider might inform the consumer, the second scenario describes an implicit drop. Implicit drops can be detected by heartbeats and timeouts.

### 6.3.5. Multi-Level Scheduling

Scheduling with cache lists and multi-hop scheduling allows to decentralize task allocation and reduces the workload of the central brokers. However, the approaches only work well when a suitable provider for the execution of the Tasklet can be found. As consumers might specify restrictive criteria for such a provider, it might happen that neither the consumer itself nor the provider which forwards the Tasklet are successful.

We introduce multi-level scheduling that follows the idea of a Rendezvous Node Tree of Kim *et al.* [193]. In such a tree, nodes send resource requests to the next higher level in a resource tree. If the parent node on this level cannot schedule the task, it passes the request to the next higher level in the tree. In our multi-level scheduling approach, we first try to use local cache lists, second the central broker of this resource pool, and finally send the request also to the brokers of other resource pools. These steps are illustrated in Figure 6.16 and discussed in the following.

① **Peer-to-Peer Scheduling:** Peer-to-peer scheduling uses cache lists held by consumers and providers. Providers might forward the request when they cannot execute the Tasklet themselves. Except for the distribution of the cache lists, no further communication between consumers and the broker is necessary.

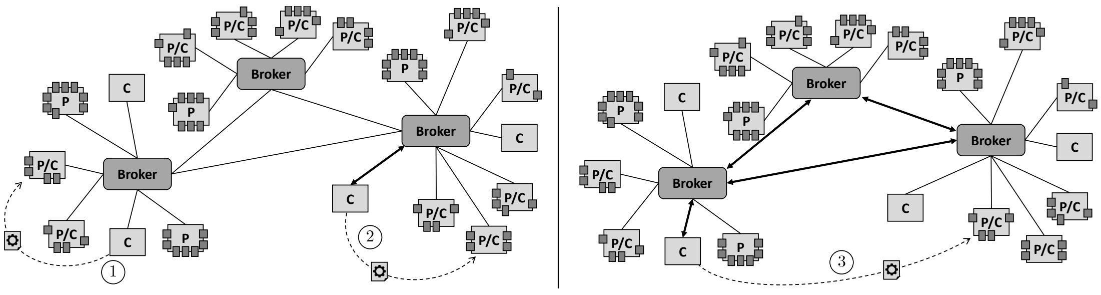


Figure 6.16.: Multi-level scheduling. In peer-to-peer scheduling (1), the consumer selects a provider from its local cache list. In intra-broker scheduling (2), the consumer requests a resource from its central broker. In inter-broker scheduling (3), the resource request is forwarded to all brokers.

② **Intra-Broker Scheduling:** In case peer-to-peer scheduling fails and even multi-hop scheduling cannot find a suitable provider, a centralized scheduling decision can serve as fallback. The consumer sends a resource request to the central broker which selects a provider. As the broker has a global view on all resources, it can even respond to highly selective resource requests. Intra-broker scheduling might take longer as peer-to-peer scheduling, as for each Tasklet execution, communication between the consumer and the broker is required.

③ **Inter-Broker Scheduling:** If even the central broker in a resource pool cannot find a suitable provider, it forwards the request to all other brokers in the Tasklet system. The brokers are connected in a peer-to-peer overlay network and exchange resource requests. Hereby, all resources that are registered in the system at that time can be reached. As inter-broker scheduling requires not only communication between the consumer and the broker but also between multiple brokers, it is only used when the previous attempts failed.

### 6.3.6. Evaluation

We evaluate the decentralized scheduling approach in multiple scenarios. The guiding question during this evaluation is whether the decentralized scheduling approach can outperform a centralized scheduler. We define the scheduling performance as the time that is required to select a provider and to send the Tasklet to this provider. We do not evaluate the reliability of the central broker nor do we test whether the broker represents a performance bottleneck when the number of requests increases. These characteristics mainly depend on the

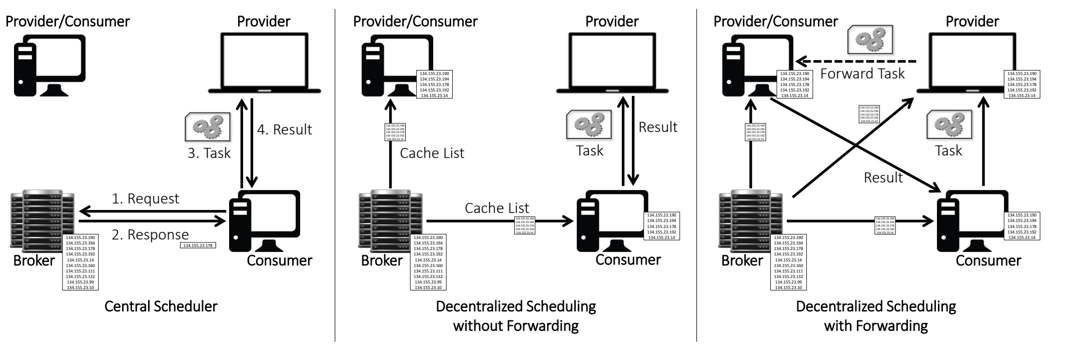


Figure 6.17.: Three scheduling strategies. A central scheduler has a global view and produces the least scheduling fails. Scheduling with cache lists leads to a delay of one round trip time when the provider is busy or offline. Multi-hop scheduling can reduce this delay by forwarding Tasklet requests.

deployed hardware and are therefore not of interest in our analysis. Instead we evaluate those scheduling parameters that can be varied to tune the decentralized scheduling approach.

The main advantage of decentralized scheduling with cache lists is that consumers can locally select a provider instead of requesting a resource from the central broker. Ideally, this saves one round trip time from the consumer to the broker. The central scheduler, in turn, has the advantage that is has a global view on the system and knows which providers are active at that time and have idle resources. Scheduling fails are rare and do only happen when the broker selects a provider before it notices that the provider has already left the system. A decentralized scheduler might send a Tasklet to a provider that has either left the system or is busy executing other Tasklets. Thus, scheduling fails are more frequent in a decentralized scheduling system. A scheduling fail results in a delay of a round trip time between the consumer and the providers. A busy provider can reduce this delay by forwarding the Tasklet to another provider. This is what we refer to as multi-hop scheduling. Figure 6.17 summarizes the three scheduling approaches.

### Evaluation Setup

For the evaluation, we implemented the Tasklet system in the discrete event simulator OMNeT++ (version 5.4.1). We used a simulated system instead of a real world testbed to be able to control all parameters and to scale the testbed to a large number of providers and consumers. In such a complex distributed computing environment, there are multiple parameters that have an effect on the

| **Parameter** | **Scenario 1** | **Scenario 2** | **Scenario 3** |
|---|---|---|---|
| **#Providers** | 1000 | 1000 | 1000 |
| **#Consumers** | 1500 | 1500 | 1500 |
| **Environment** | Stable | Unstable | Unstable |
| **Utilization** | 26% – 90% | 17% – 89% | 68% |
| **Cache List Size** | 100% | 100% | 10% – 100% |
| **Cache List Interval** | $20s$ | $20s$ | $20 - 100s$ |

Table 6.3.: Overview of the three evaluation scenarios. In each scenario all three scheduling strategies are evaluated.

scheduling performance. Among them are the number of providers and consumers, the complexity and frequency of Tasklets, the mean and standard deviation of session times that providers remain in the system, the network delay between all entities, the heartbeat rate from providers to the broker, and the timeout when the broker recognizes the leave of a provider. Parameter studies with these and several other parameters would lead to thousands of possible combinations and would exceed the scope of this evaluation. Instead, we decided to leave most parameters fixed and only vary one parameter at a time. In total, we evaluate central scheduling, decentralized scheduling with cache lists, and multi-hop scheduling in the following three scenarios (see Table 6.3).

**Scenario 1 - Stable Environment:** Providers do not leave the system and will not stop the execution of a Tasklet at any time. Scheduling fails only happen when providers are already busy executing Tasklets and have no idle virtual machines. As providers are always active, they can always forward the Tasklet when they are not able to execute it themselves. In each run, we increase the computational complexity of Tasklets which means that the load in the system gets higher. As a result, scheduling fails in the distributed scheduler become more likely.

**Scenario 2 - Unstable Environment:** Providers now enter and leave the system randomly with a session time between 5 and 10 minutes. Tasklet executions that are running during the leave are silently dropped and have to be restarted. Scheduling fails in the decentralized scheduling now also happen when the consumer sends a Tasklet to an inactive provider. As providers are offline for 50 percent of the time, only half of the resources are available and the system utilization is higher than in the stable environment. We vary the complexity of the Tasklets and compare the scheduling performance of the three approaches.

**Scenario 3 - Message Overhead:** For scheduling with cache lists, the broker has to disseminate the lists to each consumer. Further, when multi-hop scheduling is activated, the cache list also has to be sent to all providers. Thus, the overhead to distribute the cache lists grows with the number of consumers and providers and puts a considerable workload at the broker. This workload directly depends on two parameters, the cache list size and the cache list update interval. In the first two scenarios, we have set these two parameters to default values. Now, we vary these parameters to reduce the workload for the broker and observe the performance of the decentralized scheduling approaches.

### Network Delay

The performance of the scheduling strategies directly depends on the network delay between the entities in the system. The higher the round trip time between a consumer and a broker, the higher the benefit when the resource request can be avoided. A high latency between a consumer and a provider makes scheduling fails expensive and promotes central scheduling. Since the latencies have a great impact on the evaluation results, the configuration on the network parameters in the simulator requires special attention. To get an understanding of round trip times between two devices in the real world, we ran an experiment and measured round trip times between actual devices. We therefore installed a lightweight Java echo server on four Amazon EC2 cloud instances and one office computer. The four cloud servers were based in Ireland, North Virginia (US East Coast), North California (US West Coast), and Singapore. The office computer was located in Mannheim. A home-based client computer, which was also located in Mannheim, sent 250 probes with a message size of 1000 characters to each server which instantly echoed the messages. The client measured the time between sending and receiving the messages. Figure 6.18 shows the results of these measurements.

Based on these results, we determine the latencies between consumers, providers, and the central broker. We follow two assumptions. First, we argue that brokers might - in contrast to consumers and providers - be placed on well connected devices. Therefore, the average latency between a consumer and a broker is lower than the average latency between a consumer and provider. Second, as the devices in the Tasklet system might not be as well connected as the Amazon EC2 cloud instances, we assume an average latency that is slightly higher than the results

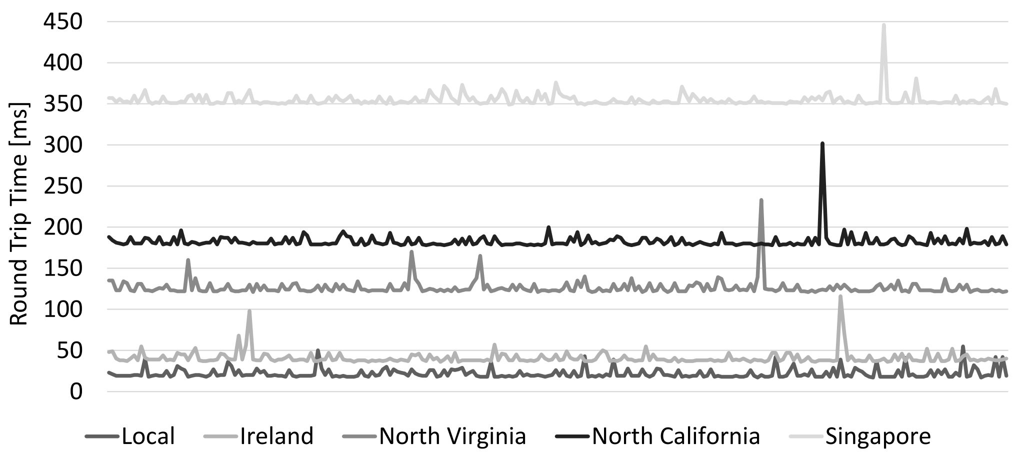


Figure 6.18.: Round trip times between a home-based client computer in Mannheim and remote servers. Each data point represents one probe with a message size of 1000 characters that was instantly returned.

from the measurements. For each combination of consumer, provider, and broker, we compute the delay based on the entities' position on a virtual map. The delay between a consumer or provider and a broker ranges from 75 to 150 milliseconds. Consumers and providers experience a delay of 100 to 250 milliseconds.

We acknowledge that these values might be different for each real-world Tasklet environment. However, this does not conflict with our findings in this evaluation but requires further simulations with adjusted parameters.

### Results of Scenario 1 - Stable Environment

We run all three scheduling strategies in a stable environment and measure the scheduling time. The scheduling time is defined as the time difference between the beginning of the first scheduling attempt and the arrival of the Tasklet at a provider that will execute the Tasklet. This means that each unsuccessful scheduling attempt increases the scheduling time. We increase the complexity of Tasklets which leads to a higher utilization of the overall system. The higher the utilization, the more likely it is that a consumer selects a provider that has no more idle virtual machines. As a result, the scheduling time increases.

The results in Figure 6.19 support this line of argument. The central scheduler at the broker performs equally well for each load. As long as there are idle providers available, the scheduler needs a constant time for the decision. This is as expected as the broker knows which providers have idle virtual machines and the

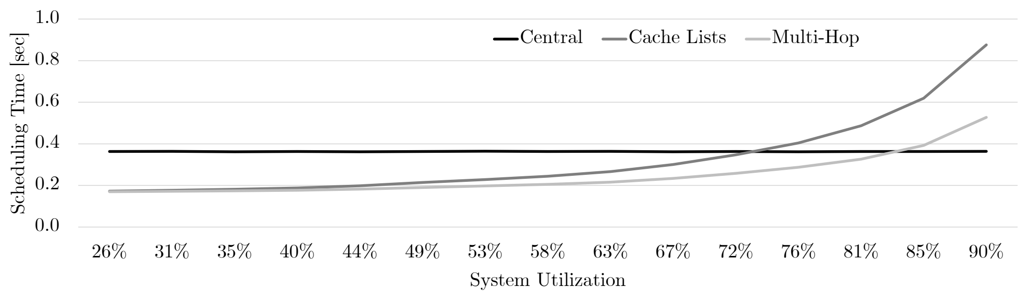


Figure 6.19.: Scheduling times for all three strategies in a stable environment (Scenario 1). The central scheduler performs equally well for all levels of utilization. The decentralized strategies outperform the central scheduler in underutilized environments but perform worse in highly utilized systems.

providers do not terminate the execution of a Tasklet. For the two decentralized scheduling strategies, however, the utilization of the system does have an impact on the performance. In an underutilized system, both strategies outperform the central scheduler. When the load in the system increases, the scheduling time for the decentralized scheduling strategies also increases. At some point, the higher number of scheduling fails cancel out the advantage of the decentralized strategies and the central scheduler starts to perform better. For multi-hop scheduling, this switch happens later compared to cache list scheduling without forwarding. This confirms the hypothesis that Tasklet forwarding increases the performance of decentralized scheduling.

### Results of Scenario 2 - Unstable Environment

In the second scenario, providers enter and leave the system at any time. As a result, the overall capacity of the system is cut in half. In addition, scheduling failures occur when consumers send Tasklets to inactive providers. This further increases the scheduling time. Providers will also drop a Tasklet execution when they leave the system. Consumers then have to restart the Tasklet and another scheduling time begins. The results in Figure 6.20 resemble those of the previous scenario. Both decentralized scheduling strategies outperform the central scheduler in underutilized environments but perform worse when idle providers become rare. In general, the scheduling time of the decentralized strategies is higher than in the stable environment as more Tasklets have to be rescheduled.

The scheduling time of the central scheduler is almost equal to the scheduling time in the stable environment (.363 seconds in stable versus .366 in unstable

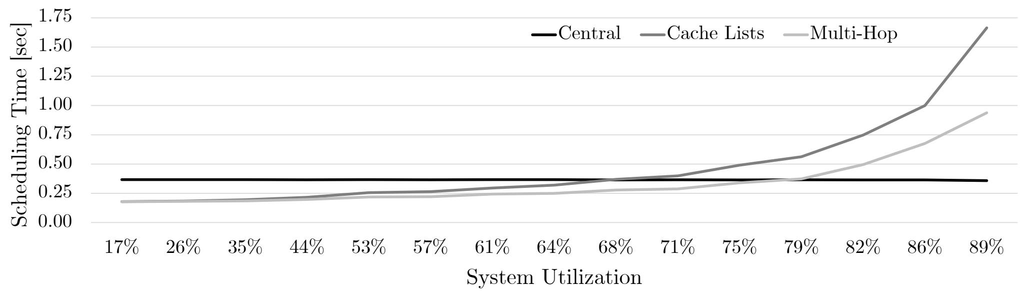


Figure 6.20.: Scheduling times for all three strategies in an unstable environment (Scenario 2). The results show that even in an unstable environment the decentralized strategies perform better than the central scheduler for moderately utilized systems.

environments) as the broker has global knowledge and knows which providers are online and idle. The slight difference can be explained by a small number of Tasklets that need to be rescheduled when the selected provider leaves the system just before it receives the Tasklet execution request. The results show that the decentralized scheduling strategies also perform well in unstable environments when the overall utilization of the system is moderate.

### Results of Scenario 3 - Message Overhead

The decentralized scheduling strategies have turned out to work well in stable and unstable environments. So far, we have only considered the scheduling time as performance measure. However, a comprehensive comparison of the three strategies requires an analysis of their message overhead. Here, we do not consider the heartbeat messages from the providers to the broker as these are identical for all strategies. Instead, we compare the amount of messages that are required for the resource requests in the central scheduling with the number of cache lists that the broker has to forward to consumers and providers in decentralized scheduling. To reduce the message overhead for the decentralized strategies, we vary two parameters: the cache list update interval and the size of the cache lists.

Figure 6.21 (left) shows the message overhead for different update intervals from 100 seconds to 20 seconds which has been the default setting for all previous measurements. The black line shows the number of resource requests that the broker has to respond to in the centralized strategy. For long update intervals the decentralized strategies require only one fourth of the messages compared to the centralized approach. The smaller the update interval, the higher the number

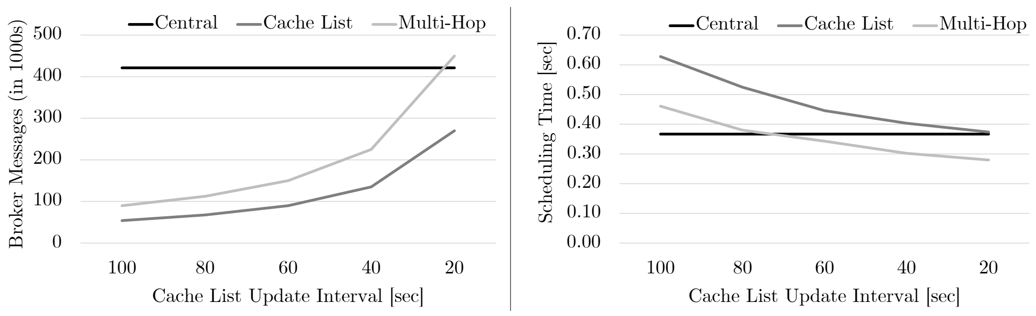


Figure 6.21.: Scheduling overhead for different cache list update intervals (Scenario 3) under a system utilization of 68%.

of cache list updates becomes. The update interval has a direct impact on the performance of the strategies (see Figure 6.21 (right)). The results only show the performance of all strategies exemplary for a system utilization of 68% to maintain readability.

For other utilizations the plots look similar but the point where the central scheduler performs better than the decentralized strategies changes. The higher the utilization, the worse is the performance of the decentralized scheduling strategies for a given update interval. For the selected scenario with a system utilization of 68%, multi-hop scheduling with a cache list update interval of 60 seconds performs better than the central scheduler and requires only about half the number of messages.

While the number of messages for the decentralized strategies can be much lower than for the centralized scheduling the size of these messages varies significantly. A response to a resource request only includes information of a single provider. In contrast, a cache list stores information about all providers in the system. To reduce the amount of data that the broker has to send, we reduce the size of the cache lists and check whether it has an impact on the scheduling performance. Therefore, the central broker only adds a given subset of active providers to the cache lists. The results in Figure 6.22 indicate that a cache list of only 10% of the active providers is sufficient to achieve the same results as with the complete cache lists for a system with a utilization of 68%. We have observed similar results for other utilization values which allows us to generalize this finding. As a consequence, the broker can reduce the message sizes by 90% without affecting the performance of the decentralized scheduling strategies.

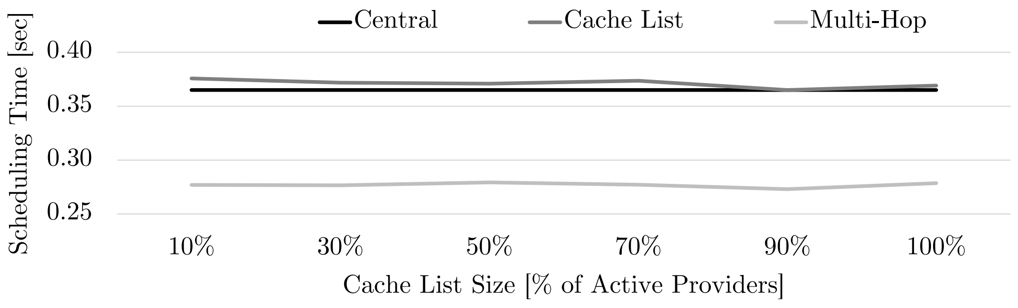


Figure 6.22.: Scheduling times for different cache list sizes (Scenario 3). The results show that the performance of the decentralized strategies does not depend on the size of the cache lists.

### 6.3.7. Discussion

We have proposed two decentralized scheduling strategies for distributed computing environments. The evaluation suggests that, depending on the latencies of the communication between the devices, these decentralized strategies can improve the scheduling performance compared to a central scheduler. Consumers can perform scheduling decisions locally which might save valuable time. However, two major limitations of this study have to be considered.

First, even though the evaluation has been performed in a large scale simulation, the results depend on a large number of parameters. We have discussed some of these parameters in the beginning of the evaluation. As these parameters are different for each system, the absolute numbers in this evaluations cannot be generalized. Even though, in this evaluation, more than 140 million Tasklets were executed and more than 128 gigabytes of log data were analyzed, only a small subset of possible combinations could be tested. To actually determine the perfect settings for a scheduler in a distributed system, further simulations have to be performed when the parameters of this system are known.

Second, the benefit of decentralized scheduling is limited to about one round trip time between consumers and the broker. While this time span varies for each system, it typically does not take longer than several hundreds of milliseconds. Thus, application areas for decentralized scheduling include highly responsive systems with a large number of sub-second tasks. For more complex tasks that require tens of seconds or minutes of execution time, the reduced scheduling time might not be of great importance. However, these systems might benefit from a reduced number of messages and a relaxed dependency of the broker as single point of failure. Decentralized scheduling increases fault-tolerance as scheduling decisions can be made even if the central entity is temporarily offline.

## 6.4. Further Scheduling Features

So far, we have introduced three optimizations for the Tasklet system that have resulted in major design and implementation changes. Besides these substantial changes, we have implemented and evaluated three further extensions to the Tasklet middleware that we will discuss in the following.

### 6.4.1. Priority Tasklets

Our current implementation of the Tasklet system does not support queuing of Tasklets and requests to busy providers are rejected. However, supporting queues would allow for flexibility as it decouples the scheduling process from the execution. While queuing does not solve the problem of a permanently congested execution environment, it makes a system tolerant to temporary high loads. Tasklets that are scheduled during a high load are not rejected but added to a queue and will eventually be executed. On the one hand, this reduces the workload for the scheduler. On the other hand, it introduces the risk that Tasklets are delayed as they are queued for some time. Thus, in this section, we introduce priority Tasklets that can skip the lines and are executed before any non-prioritized Tasklet [208][5].

#### Implementation

To implement priority Tasklets we used the concept of QoC. Priority can be either implemented as a QoC goal which developers can specify for a Tasklet or a QoC mechanism that enforces one or more QoC goals. As priority is not a goal per se but rather facilitates a timely execution and avoids delays, we implemented it as a mechanism that can be used to enforce the speed goal. To avoid the excessive use of the *Priority* QoC mechanism, the usage might be charged.

[5] [208] is joint work with S. VanSyckel, C. Krupitzer, JM Paluska, and C. Becker

In a Tasklet system that supports queuing, resource providers execute Tasklets in a FIFO order. One Tasklet is executed at one TVM at a time. Hence, the waiting period for a Tasklet is determined by the workload of the provider it has been assigned to.

Introducing a simple model of execution levels, we distinguish between two different queues. In order to execute the incoming Tasklets based on their priority declaration, the modification of the providers is minimal. Instead of holding only one queue for Tasklets, each provider holds two queues – the *economy queue* and the *priority queue*. Having completed the execution of a Tasklet, the provider first checks the priority queue for further assignments. Only if the priority queue is empty, the provider checks and processes its economy queue. Hence, as long as a provider has prioritized Tasklets in its queue, the economy Tasklets are neglected. In case both queues are empty, the provider will proceed with the next incoming Tasklet, whether priority or not. As there are no guarantees for the best-effort Tasklets, priority execution does not guarantee an immediate execution or a maximum waiting period either, but ensures that each priority Tasklet is processed before the next economy Tasklet will be executed.

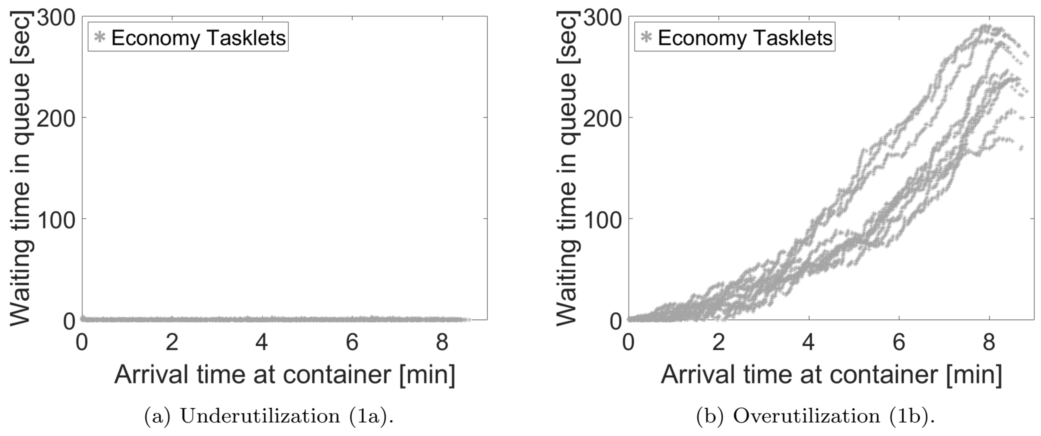
Developers do not directly request Tasklets to have priority as they cannot know whether Tasklets are executed in a congested environment or in an environment where multiple idle resources are available. Thus, developers only set the *Speed* QoC goal for time-critical Tasklets. The Tasklet middleware then evaluates the current state of the system and decides how this goal should be enforced. In Section 6.1, we have already demonstrated how the QoC mechanisms *Strong Distribution* and *Speed Filter* can be used to reduce the execution time. When the middleware detects that the system is congested, it can use *Priority* as a (further) mechanism to implement a timely execution. In case of an underutilized system, this mechanism might not be used to avoid the charges for prioritized Tasklets.

### Evaluation

We evaluated priority Tasklets on two computers and a blade server connected via Ethernet. The applications that created the Tasklets were executed on a Lenovo T430s laptop equipped with a 64-Bit Windows 7 Professional, an Intel i7-3520 Dual-Core 2.9 GHz CPU, and 8 GB RAM. For the providers we used a desktop PC running a 64-Bit Windows 8 OS equipped with an Intel i5-2500K Quad-Core 3.3

| Scenario | 1a | 1b | 2a | 2b | 3 |
|---|---|---|---|---|---|
| # of applications | 20 | 20 | 20 | 20 | 10 + 10 after 120s |
| # of Tasklets per application | 100 | 200 | 100 | 200 | 200 / 200 |
| # of providers | 10 | 10 | 10 | 10 | 10 |
| Seconds between requests | 2-8 | 1-4 | 2-8 | 1-4 | 3-9 / 1-3 |
| Economy/priority ratio | 100/0 | 100/0 | 80/20 | 80/20 | 80/20 |

Table 6.4.: Overview of evaluation scenarios.



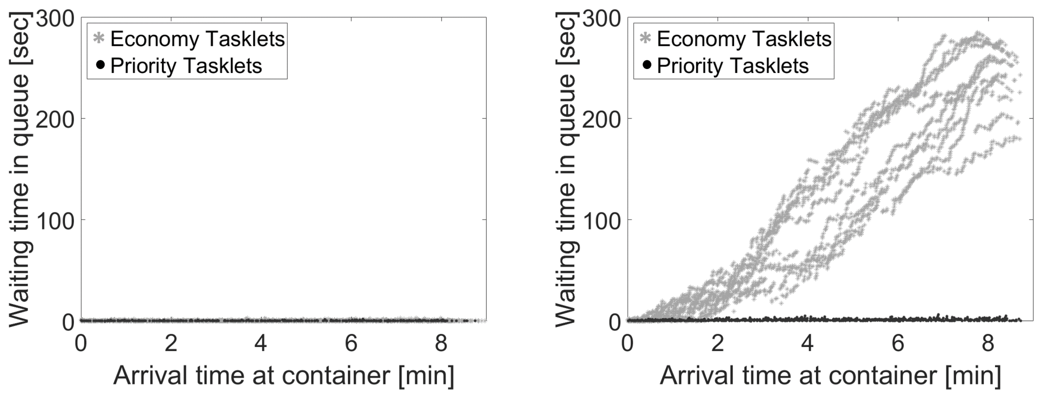
(a) Underutilization (1a). (b) Overutilization (1b).

Figure 6.23.: Waiting period per Tasklet in Scenario 1a and 1b.

GHz CPU and 8 GB RAM. Finally, the single broker ran on the blade server with a 2x Quad-Core Intel Xeon with 2.33 GHz CPU, 6 GB RAM, and a 64-Bit Windows Server 2008 Standard Edition. During the evaluation, the applications repeatedly issued Tasklet execution requests to the broker based on the parameters of the individual scenario. For each scenario, we specified the number of applications in the system, the number of Tasklets per application, the number of providers, the frequency of Tasklet requests and the ratio between priority and non-priority (economy) Tasklets. Table 6.4 summarizes these scenarios.

The first two scenarios (1a, 1b) served as a benchmark and only involved economy Tasklets, meaning that all Tasklets had the same priority. Providers strictly stuck to the FIFO order for Tasklet execution and there was no way to avoid a delayed execution in case of a congested system.

In the second two scenarios (2a, 2b), we used the same basic settings as before, but labeled 20% of the Tasklets as priority Tasklets. In both, Scenario 1a and 2a, we underutilized the capacity of the providers in order to examine the difference in waiting time between economy and priority Tasklets. This means that the



(a) Underutilization (2a). (b) Overutilization (2b).

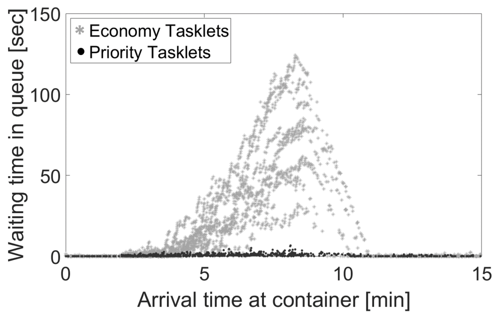
Figure 6.24.: Waiting period per Tasklet in Scenario 2a and 2b.

providers could execute the Tasklets faster than they came in. In scenarios 1b and 2b, in contrast, we increased the number of Tasklets per application and reduced the waiting period between Tasklet execution requests, in order to overutilize the providers. Here, the waiting time of the economy Tasklets was expected to increase.

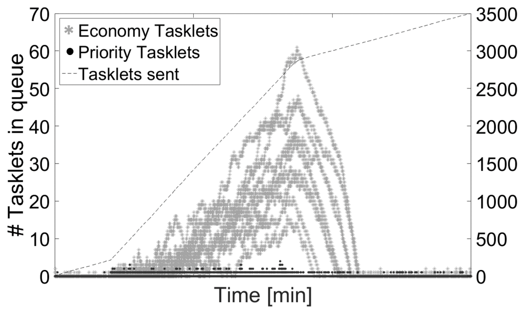
Scenario 3 featured both under- and overutilization in order to see how the system recovers from overload. That is, we created a steady workload that underutilizes the providers comparable to Scenario 2a. After two minutes, we started a second set of applications and flooded the providers with Tasklet execution requests, in order to create a peak in the workload. When the second set of applications has finished sending Tasklets, we expected the providers to slowly work off the build-up. Even during peak load, we expected priority Tasklets to be executed with a minimal waiting period.

Figure 6.23 shows the waiting period of the Tasklets in Scenarios 1a and 1b during underutilization (Figure 6.23a) and overutilization (Figure 6.23b). Whereas in the first case, the Tasklet execution was almost immediate, the waiting period increases linearly when the queue for economy Tasklets built up in size during overutilization.

Similar to above, Figure 6.24 shows the waiting periods of the Tasklets in Scenario 2a and 2b. During underutilization (Figure 6.24a) Tasklet execution was almost immediate, and there was no measurable difference in the waiting periods of the economy and priority Tasklets. During overutilization (Figure 6.24b) we see that



(a) Waiting period per Tasklet.



(b) Queue sizes per Tasklet.

Figure 6.25.: Waiting time and queue size in Scenario 3 in which the system is temporarily overloaded.

even though the providers could not keep pace with their economy queue, the waiting time for priority Tasklets stayed consistently small as intended. Naturally, this would no longer be the case when the amount of priority Tasklets exceeded the provider capacities.

Figure 6.25a shows the waiting period per Tasklet in Scenario 3. At first, the providers were not fully utilized and so the waiting periods for both economy and priority Tasklets were very short. After two minutes, the second set of applications was launched and the number of Tasklet execution requests increased enough to exceed the computing capacity of the providers. However, as in Scenario 2b, the priority Tasklets were still executed almost instantly, whereas the economy Tasklets' waiting periods increased dramatically and peaked at about the eight minute mark, at which the second set of applications had finished requesting Tasklet executions (see Figure 6.25b). Following this peak, the system recovered and worked off the build-up.

Figure 6.25b shows the respective queue sizes with regard to the accumulative number of requests (dashed line) in Scenario 3. The behavior of the queue sizes was consistent with the waiting periods shown in Figure 6.25a. The size of the economy queue increased dramatically with the start of the second set of applications, while the size of the priority queue stayed almost constant. Overall, our evaluation shows that priority Tasklets were always executed after a consistently short delay, even if the overall workload exceeded the capacity of the providers.

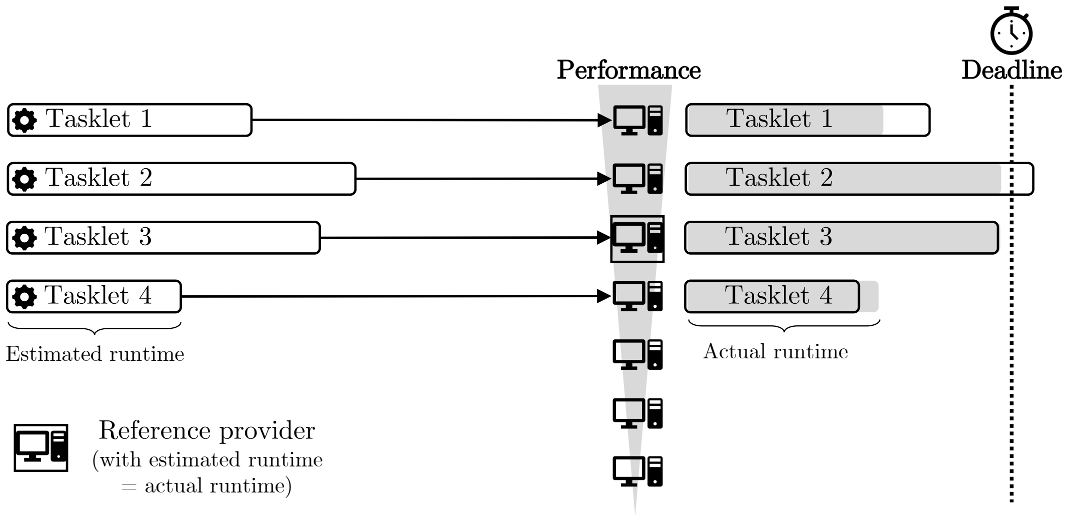


Figure 6.26.: Greedy scheduling in a moderately utilized environment. All Tasklets can be computed within the deadline.

### 6.4.2. Resource Reservation

In the current implementation of the Tasklet system, the execution of each single Tasklet can be optimized individually. There are no scheduling strategies that consider the interdependencies between multiple Tasklets. This fits in with the general concept of Tasklets which are self-contained, independent units of computation.

However, we argue that there are applications that might benefit from a coordinated Tasklet scheduling strategy which optimizes on the level of an entire application [209, 210][6][7]. Therefore, we discuss an application that issues multiple Tasklets that have different computational complexities but all share a similar deadline. This means that a Tasklet must be computed within, for example, 2 seconds regardless of its complexity. In a computing environment that consists of heterogeneous resource providers, it might happen that some devices are fast enough to compute even complex Tasklets in time whereas others fail to meet the deadline. Thus, an optimal scheduler should always select the fastest resource provider for execution.

[6] [209] is joint work with M. Pfannemüller, M. Weckesser, R. Kluge, M. Luthra, R. Klose, C. Becker, and A. Schürr

[7] [210] is joint work with M. Pfannemüller, M. Weckesser, R. Kluge, M. Luthra, R. Klose, C. Becker, and A. Schürr

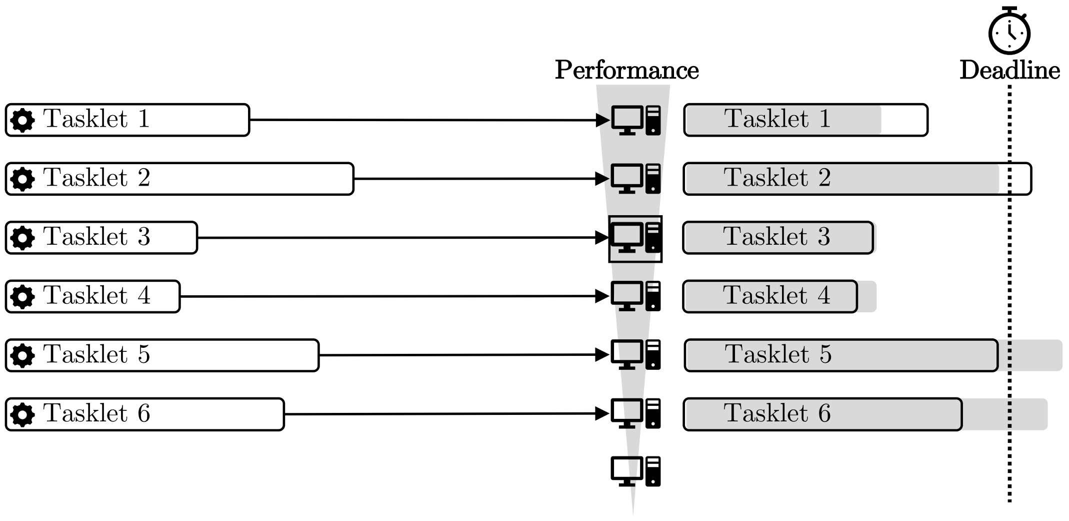


Figure 6.27.: Greedy scheduling in a moderately highly environment. Tasklets 5 and 6 fail to meet the deadline.

Figure 6.26 shows this scenario. An application issues 4 Tasklets at almost the same time. The Tasklets are sequentially scheduled to the most powerful idle resource provider. The runtime of a Tasklet can be estimated by its complexity and the performance of a given reference provider (3rd from top). A Tasklet that gets scheduled to a more powerful device than the reference provider will be executed in less time than the estimated runtime (compare Tasket 1 and Tasklet 2) and vice versa (compare Tasklet 4).

In a moderately utilized environment, the *greedy* strategy in Figure 6.26, which always selects the fastest available resource provider, works well and successfully meets all deadlines. However, when the load of the system gets higher, there is an increasing risk that complex Tasklets are assigned to slow resource providers, because fast providers are already used. Figure 6.27 shows how the greedy strategy in a highly utilized environment cannot meet all Tasklet deadlines anymore (compare Tasklet 5 and Tasklet 6). In this example, less complex Tasklets (3 and 4) are scheduled to fast providers even though they could have been executed on slower providers within their deadline.

### Implementation

To avoid deadline misses in highly utilized environments, we introduce an alternative scheduling strategy that reserves powerful providers for complex Tasklets.

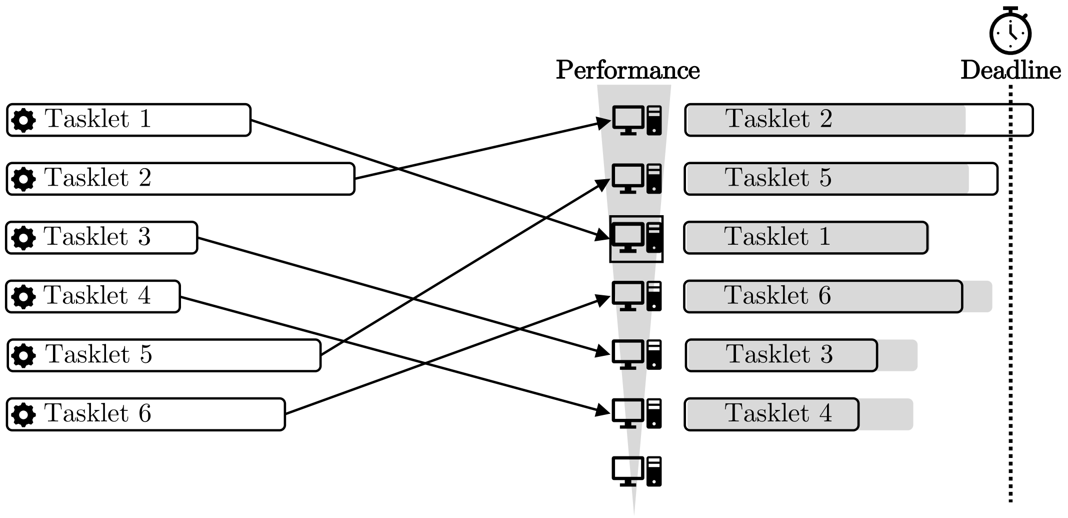


Figure 6.28.: Fuzzy scheduling in a highly utilized environment. All Tasklets can be executed in time.

Less complex Tasklets are scheduled to slower providers. This strategy leads to an overall lower performance as powerful resources remain idle while they are waiting for complex Tasklets. However, the strategy reduces the number of deadline misses. As the scheduling decision is not as straightforward as in the *greedy* scheduler above, we call the alternative strategy *fuzzy*. Figure 6.28 shows an example of how the *fuzzy* strategy schedules Tasklets in a highly utilized environment. The *fuzzy* strategy is based on the assumption that the complexity of Tasklets can be estimated.

The scheduling decisions in the *fuzzy* strategy depend on multiple parameters that determine, for example, if a Tasklet is considered as complex or as easy to compute. Further, the resources have to be categorized into slow and fast providers. In general, Tasklet complexity and provider performance can be categorized into two, three, or multiple levels. The number of levels and the thresholds between them must be defined based on the properties of the Tasklets and resources in the environment.

## Evaluation

We evaluated the fuzzy strategy in the OMNeT++ Tasklet simulator that we have already used to evaluate our decentralized scheduling strategies in Section 6.3. We categorized the complexity of our Tasklets into three groups, *low*, *medium*, and

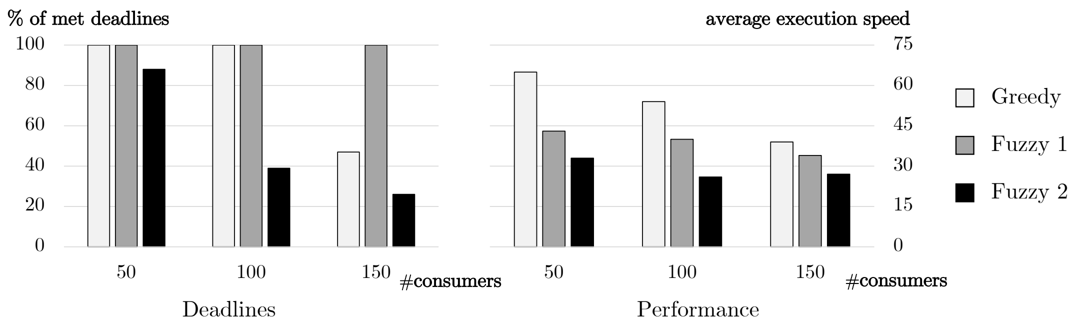


Figure 6.29.: Results of the greedy and two fuzzy scheduling strategies. The greedy strategy performs best in low and moderately utilized environments (50 - 100 consumers) but results in multiple deadline misses in highly utilized environments. The performance of the fuzzy strategies highly depends on their parametrization.

*high*. Therefore, we normalized the complexity to a range between 0 and 100 where $T_{low}$ is the threshold between *low* and *medium* and $T_{high}$ is the threshold between *medium* and *high*, respectively. We also split up the provider pool into three groups, *slow*, *medium*, *fast*. The percentage of slow providers is defined as $P_{slow}$ and the percentage of fast providers as $P_{fast}$. The remaining providers were categorized as *medium*. We set up the evaluation with 150 providers and three different scenarios with 50 consumers (low utilization), 100 consumers (moderate utilization), and 150 consumers (high utilization). We evaluated the *greedy* strategy against two *fuzzy* strategies, *Fuzzy 1* with $(T_{low} = 20, T_{high} = 60, P_{slow} = 33, P_{fast} = 33)$ and *Fuzzy 2* with $(T_{low} = 20, T_{high} = 80, P_{slow} = 25, P_{fast} = 33)$. These values were identified empirically and are used to demonstrate the importance of a careful parametrization of the *fuzzy* scheduling strategy.

The results in Figure 6.29 show the performances of the three strategies. The *greedy* scheduler always shows the highest average execution speed and performs well in low and moderately utilized environments with 50 and 100 consumers. However, when the number of consumers and, thus, the number of Tasklets in the system increased, the scheduler could not meet all deadlines anymore. In contrast, the *Fuzzy 1* strategy met all deadlines even in the highly utilized environment. It also showed a relatively high execution speed. The *Fuzzy 2* strategy, however, failed to meet deadlines even in underutilized environments (50 consumers) and showed an overall low execution speed.

The evaluation above is not meant to determine the optimal parameters for the *fuzzy* scheduling strategy in the Tasklet system. Rather, it shows how sensitive the performance of the system is to the parameters of the strategy. Non-optimal parameters can lead to worse results than following the *greedy* strategy and reduce the overall performance of the system. Thus, the schedulers in each individual Tasklet environment must monitor the context of the environment and set these parameters dynamically. This requires a thorough analysis of the interdependencies of the parameters, available providers, existing consumers, and type of Tasklets. This analysis is beyond the scope of this thesis.

### 6.4.3. Demand Forecasting

In the current implementation of the Tasklet system, each Tasklet execution requires a resource request from the consumer to the broker and a resource response from the broker back to the consumer. This approach is feasible for applications that spontaneously offload workload to remote resources. However, in stream processes, where work packets are offloaded continuously, requesting a resource for every Tasklet will lead to a considerable communication overhead and unacceptable delays. To reduce both, we propose a demand estimation mechanism that predicts the future demand of stream processes [160][8]. Thus, we can supply each process with a number of resources even before tasks are created. A feedback mechanism from the process helps to make the prediction more accurate. This is an essential part of the mechanism as overestimating the demand would lead to resources that are reserved but would not be used. Whereas the prediction is straightforward in processes with a uniform workload, it gets nontrivial for event-based processes such as in [211] where a face recognition task is issued as soon as an infrared sensor detects a person moving into a security zone. The occurrence of such an event is randomly distributed and can be estimated by fitting a probability distribution function to observed data.

#### Implementation

Predicting future workload for stream processes is, for example, discussed in [212] and [213]. Future workload can be estimated by using fundamental knowledge of statistical distribution and time series analysis. It is based on two main parts: (i) fitting the distribution to a historical workload and (ii) calculating the

[8] [160] is joint work with , S. Choochotkaew, H. Yamaguchi, T. Higashino, D. Schäfer, and C. Becker

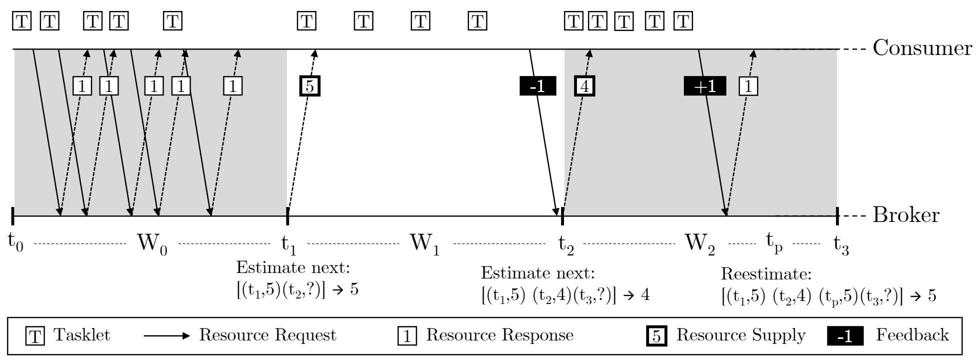


Figure 6.30.: Demand forecasting mechanism. For each window W the resource demand is predicted based on the demand in the previous windows.

future workload by a time series analysis [213]. We propose a demand estimation mechanism for stream processes that splits up the timeline into evenly sized windows. For each window $W$, we count the number of tasks and use this data to predict future demands. Figure 6.30 shows the demand estimation mechanism. In the initial window ($W_0$), the broker performs the task allocation on a per-request basis and counts the number of requests. It fits a probability distribution function (PDF) to the data which is used for future resource prediction. As the tasks appear at random points in time, we have selected a Poisson distribution, estimate the parameter $\lambda$, and determine the goodness of fit with chi-square ($\chi^2$). In the beginning of the next window ($W_1$), the broker sends the number of resources proactively to the consumer. In this way, it does not only reduce the delay but also eliminates the need for a resource request. In case not all resources are used at the end of a window, the consumer sends a feedback to the broker. In case a consumer needs more resources than it received from the broker (as in $W_2$), it also sends a feedback to the broker indicating that it requires additional resources. The feedback messages help the broker to learn about the request pattern of the broker. After each window, the PDF is updated.

In general, there is a trade-off between sending too few or too many resources to the consumer. If the number of required resources is underestimated, the consumer has to send another resource request which will cause additional delay and communication overhead. If the number of required resources is overestimated, the resources remain unused as they are reserved for one particular consumer.

We note that in some cases the number of requests may be not be significant and using the estimation may incur unnecessary overhead. For such a case, we

| Application | Task Arrival Pattern |
|---|---|
| TRACK | Every time when a person enters the common area, based on the data of a double passive infrared sensor (PIR) |
| ENV0 | When the temperature is out of range from 20°C to 30°C. |
| ENV1 | When the temperature is out of range from 20°C to 30°C and the humidity is out of range from 40% to 60%. |
| ENV2 | When the average temperature of one hour is out of range from 20°C to 30°C. |
| ENV3 | When the average temperature and humidity in one hour is out of range from 20°C to 30°C and from 40% to 60%, respectively. |
| VIDEO | We feed uniform requests every 100s based on camera sensors. |

Table 6.5.: Applications used in the evaluation and their task arrival patterns.

additionally specify the minimum threshold, $N_wTH$, to determine whether the broker should preallocate resources or not by considering the trade-off between feedback-message overhead plus wasteful allocation and request-message frequency. If the calculated number of resources from the currently considered window is less than $N_wTH$, the broker will allocate one virtual machine responding to each request only, considered as an on-demand response. Otherwise, it will use the estimated response.

### Evaluation

To evaluate our approach, we conducted an experiment on collected data from a real deployment in our laboratory that provided realistic arrival patterns. There were six sampling applications that initiated a task based on the data of three sensors under specific conditions. The task arrival patterns are shown in Table 6.5. We simulated a resource sharing system written in Java and measured (i) the number of communication packets including task request and feedback count from a consumer to a broker, (ii) the response count from the broker back to the consumer, and (iii) the number of unused resources at the consumer due to overestimation. We also measured the effect of setting the minimum threshold $N_wTH$, to activate the advanced allocation. Further, we compared our Poisson-based prediction model with the trivial approach called *Lookback*, which predicts the next window to have the same demand as the previous one.

We compared the total communication costs after running 50 windows for all models against results without estimation. This is shown in Table 6.6 with a window period of $10,000$ seconds. We observe, that applying the Poisson model

| | NO EST | LOOKBACK | POIS-0 | POIS-1 | POIS-2 |
|---|---|---|---|---|---|
| **VIDEO** | 8448 | 241 | 139 | 214 | 230 |
| **TRACK** | 104 | 123 | 73 | 96 | 96 |
| **ENV0** | 264 | 107 | 119 | 147 | 195 |
| **ENV1** | 264 | 107 | 119 | 147 | 208 |
| **ENV2** | 58 | 89 | 55 | 54 | 55 |
| **ENV3** | 58 | 89 | 55 | 54 | 55 |

Table 6.6.: Total communication packets during 50 windows.
(POIS-$N$: fitting with POISSON distribution and $N_wTH = N$)

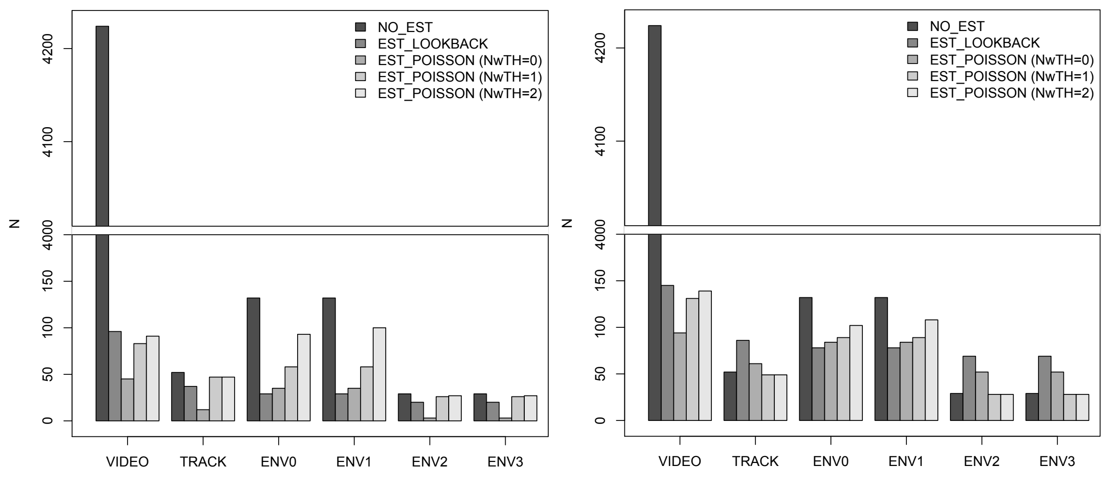


Figure 6.31.: Request (left) and response (right) count from demand estimation mechanism during 50 Windows

is always beneficial, especially when the number of requests per window is high, like with the *VIDEO* sampling stream. In general, the Poisson model reduced the communication costs and the lower minimum threshold ($N_wTH = 0$) mostly provided a better estimation. For some cases, *Lookback* estimation incurred slightly lower communication costs. However, the simplicity of this model can also cause unexpected larger costs due to estimation failures.

Figure 6.31 shows that the estimation model can reduce the request count in most cases. Although, in a uniform requesting case, such as *VIDEO*, the *Lookback* model can perform better than the more sophisticated models, but it usually suffers from false estimation when the request arrival is fluctuating, as observed in *TRACK*, *ENV2* and *ENV3*. The increased response count with periodic feedback might affect the total communication if we do not limit the minimum threshold of activation due to sparse requests. It might also be noted that if we set $N_wTH$ to 1, the response count will not be larger than the base case *NO EST*. Resource overestimation can cause wasted allocation, which keeps other consumers from using the resources. A higher threshold, $N_wTH$, can reduce the wasted allocation, especially in the low arrival sampling stream.

Overall, the evaluation results show that we can apply the pre-allocation mechanism, a proper distribution model, and threshold activation, to reduce the total communication cost, especially for highly frequent request interval applications.

# 7. Social Computing

Over the course of this thesis, we have developed a distributed computing system and have presented several extensions to optimize task scheduling. In a proof of concept implementation, we have demonstrated the feasibility of such a system that could be deployed in the real world. Resource providers and resource consumers could install the Tasklet middleware and exchange computation between their devices. While these contributions are of a purely technical nature, we have left untouched the questions of who would be using the system and why device owners would share their resources with others. However, as the success of the system relies on the acceptance and the active participation of resource providers, we have to understand their motivation to participate. Further, because computational resources are shared in a P2P manner, we have to take the social relationships between the users into account [214] and identify the reasons for them to contribute to the system. We must not neglect the fact that each device in the system is owned by persons who have their own ideas about who should be allowed to use the computational power of their device and what they expected to receive in return. Specifically, the social relationships between resource providers and resource consumers might not only determine whether resources are shared but also how the resource provider is compensated. The idea of leveraging social relationships to establish a level of trust between users and to increase the motivation to share resources is manifested in the idea of *social clouds*.

> *A Social Cloud is a resource and service sharing framework utilizing relationships established between members of a social network.* [215, p.1]

In this chapter, we discuss the relationships between providers and consumers as well as the incentives for sharing resources. In particular, we will have a look at social clouds, discuss how these relationships determine the expected compensation and finally implement a social cloud layer on top of the Tasklet system. This chapter is structured into three sections. In Section 7.1, we elaborate on incentives for device owners to participate in social computing systems. We also analyze the social relations between resource providers and resource consumers and discuss reasons that might affect the willingness to participate.

Based on these findings, in Section 7.2, we investigate the effect of monetary incentives on the willingness to share resources. We conduct a scenario experiment in which we manipulate the social relationships between consumers and providers as well as the compensation for sharing resources.

In Section 7.3, we design and implement a social network as an overlay on top of the Tasklet system. The social network allows users to establish friendship connections between each other. These relationships are then used to perform a social Tasklet scheduling.

## 7.1. Incentives and Relationships in Social Computing

Sharing of computational resources might be motivated by multiple different factors. Device owners might share for altruistic reasons or to receive some money as compensation. We argue that this motivation does not only depend on the providers themselves but also on their relationship to the resource consumers. People might rather share their computational power with friends than with strangers and even might do so for free. The success of the BOINC platform [9] where volunteers contribute to scientific projects and do not receive any compensation in return for their computational power, suggests that the willingness to share is not only driven by monetary reasons. However, it is highly questionable whether the same arguments would hold for sharing with for-profit organizations or anonymous users. In accordance with the literature, we have identified three key factors that have an impact on the users' willingness to participate in volunteer computing. These factors are *incentives*, *social relationships*, and *obstacles*. Related work has elaborated on these three aspects in an isolated manner (incentives [206, 214–231], social relationships [215, 223, 230, 232–235], obstacles [225, 231, 235–237]). To the best of our knowledge, there is no comprehensive study that analyzes these aspects in existing computational resource sharing systems. Thus, in this section, we discuss these factors and classify existing computational resource sharing systems based on their incentives schemes and the relationshipss between consumers and providers.

### 7.1.1. Incentives

To understand why people get involved in computational resource sharing we first need to understand their motivation. According to Self-Determination Theory, two basic types of motivation can be distinguished, i.e., *intrinsic* and *extrinsic* [238]. A motivation can be described as *intrinsic* when a person does something "*for its inherent satisfactions rather than for some separable consequence*" [238, p.56]. Intrinsically motivated means being driven by curiosity, fun, or challenge. Thus, an activity itself is satisfying. In contrast, extrinsic motivation relates to the expectation of an outcome that is separable from the actual activity. In the absence of this outcome, people that are extrinsically motivated would rather not participate in this activity.

Intrinsic and extrinsic motivation are neither exclusive nor immune to changes. Many actives are performed for multiple reasons and the balance between intrinsic and extrinsic motivation changes over time. A person who starts working in a job only for money, i.e., has an extrinsic motivation, might start enjoy doing this job and develop an intrinsic motivation. Further, activities that might seem to be purely intrinsically motivated, such as donating money, could be affected by extrinsic motivations such as social pressure. In [221], Dellavigna *et al.* show that social pressure is an important factor in the context of door-to-door giving that can even be monetarized. Keeping this knowledge about different types of motivation into account, we further discuss different types of incentives for computational resource sharing. We start with the most intrinsic type of incentives, altruism, and end with a purely extrinsic incentive, namely monetary compensation. We do not claim that this order is indisputable but use it to present the different types of incentives in a meaningful order.

**Altruism:** The concept of altruism is often referred to as a basic motivation in crowd sourcing scenarios [239]. Altruistic users are willing to share their resources without expecting any kind of compensation. The mere activity of sharing and the feeling of a *warm glow* are inherently enjoyable or rewarding [240]. This can for instance be the case for some scientific projects in which users support the goals of the project and are intrinsically motivated to move the idea forward [9, 214]. This sense of meaningfulness is represented by the type of projects that draw the most volunteers on the BOINC distributed computing platform [241]. Many of these non-profitable projects are either located in the area of astrophysics, chemistry, mathematics, or medicine. By sharing their resources with these projects, volunteers can contribute to science. Therefore, each project provides a description of its goals and explains why providing resources to the project results in an overall benefit for mankind. Rashid *et al.* [242] argue that displaying the value of a contribution is an important motivator for people to share resources.

Altruism does not infer that there is no benefit for the altruistic person at all. Rather, this self-benefit is purely non-material but typically results in a certain kind of satisfaction [243]. This satisfaction can either result from a self-reward and the feeling of being a good person or can avoid feelings of shame and guilt [243]. An example for the latter are airline passengers that participate in the carbon offset program of an airline to make up for the additional environmental damage they cause by flying. In the case of computation sharing, an altruistic reason to participate could be to share resources in order to increase the utilization of computational devices and, thus, reduce the overall energy consumption. While this is undoubtedly an altruistic incentive, the user has a personal long-term benefit by living in a healthier environment.

**Gamification:** Starting from 2011, the idea of gamification has become popular. Gamification describes the embedding of game elements into non-game contexts to make these experiences enjoyable [244, 245]. In volunteer computing, gamification can be found when contribution is rewarded with credits that can be compared with those of other participants [241]. Even though these rewards do not have an actual monetary value, they may increase the motivation to participate and introduce competition between users. In this rather intrinsic type of motivation, the user finds the activity of sharing enjoyable, however, it is not an inherent trait of the activity, but an added aspect of gaming or semi-gaming experience. Satisfaction and enjoyment can be considered as parts of gamification behavior that explain participation in resource sharing activities [214, 246, 247]. The potential of gamification has been shown in multiple studies [245, 248, 249].

On the individual level, gamification can be used as motivation to encourage users to achieve a certain goal. Among others, gamification elements include badges, leaderboards, levels, time constraints, and limited resources [247]. In the context of computational resource sharing, gamification can be added in multiple ways [250].

Users can be granted achievements or scores when their contribution to a scientific project is high enough and the results can be published on a leaderboard [241].

**Reputation:** Reputation becomes relevant as a motivation when achievements are connected to real persons, for example, when users of volunteer computing have the possibility to share their results in social media networks. In [218], Ariely *et al.* show that people have a strong desire to be seen by others as doing good. Communicating that they have provided computational resources, for example, for a scientific project can be a signal that individuals can use to build up a reputation for doing good. The concept of reputation is related to the theory of social comparison that describes the drive to compare a person's own opinions and abilities with those of others [251]. Nov *et al.* describe reputation as an extrinsic incentive [214]. Sharing leads to a gain of reputation among like-minded people which, in turn, enhances the willingness to share. Reputation as an incentive to share must not be mistaken for reputation as a means to establish trust in a peer-to-peer environment [252, 253]. The concept of reputation is related to the idea of gamification when, for example, ranks are rewarded to top contributors. However, in contrast to gamification, reputation has a signaling effect to enhance the status in the community [254].

**Reciprocity:** Sharing can be performed in a reciprocal way. Resource providers retrieve a credit as compensation for their shared resources. This credit can then be used to request computational power from other participants when these providers requires additional computational resources themselves. There are multiple ways to implement reciprocal sharing systems. In a one-to-one implementation, the exchange happens directly, in a *tit-for-tat* manner and the resource usage between two users needs to balance out [255]. A more flexible solution is to introduce a credit system which allows to trade computation between multiple users [256]. The most relaxed assumption is that there is no explicit accounting but computation is considered a public good. Users contribute as much as they can and, in turn, are allowed to use resources at any time and as much as they want [257]. Such systems require a high level of trust between the participants as they are prone to free-riders. Computation providers participate in the system as they require additional computation at another point in time. These systems only work well when the demands of the participants match in the long run and the same kind of resource is required.

**Virtual Currency:** Even though reciprocity can be one way to solve the free-rider problem, it is limited to scenarios in which the exact same resource is traded. This may lead to a situation in which a user who mainly wants to use computational resources from other users but cannot provide these resources in the future will be excluded from the system. In turn, users that own powerful resources accumulate credits that they will not need to spend as they are never reliant on remote computational resources. A virtual currency can help to solve this problem as it might enable users to trade computational power for other resources such as bandwidth or storage [258]. Virtual currencies are a popular near-monetary incentive. In contrast to a credit-based system, virtual currencies can implement market mechanisms such as auctions, inflation, deflation, and interest rates [252, 259, 260]. Virtual currency can be analyzed and implemented on two different levels. They can either be transparently handled by the system [230, 261, 262] or actively managed by the user [252, 263].

**Monetary:** Finally, users in computation sharing systems can even use real money in transactions. When users pay for using remote resources, they do not have the obligation to share resources themselves. Instead, each resource has its price which can be either fixed, variable, or negotiated between resource consumer and provider. Participants can be either consumers, producers, or both. Sharing resources can then be considered as a business model and the requirements for security, trust, and persistence are higher than in non-monetary systems. Accounting can be performed by a trusted third party or in distributed solutions such as blockchains. Monetary compensation can be attributed to extrinsic motivation, as the outcome of a fulfilled task is separable from the task itself [238]. Buyya *et al.* present different economic models for grid computing, including commodity markets, posted price model, bargaining, tender/contract-net model and auctions [264].

### 7.1.2. Relationships

A fact that is often neglected in distributed computing systems is that devices are owned by private users and that these users might have different types of relationships with the resource consumers. This leads to a change of perspective in terms of incentives for sharing and the level of trust between resource providers and consumers. In the following, we categorize different relationships and discuss which impact these relationships may have on the willingness to share resources.

**Anonymous:** In many distributed computing systems, resource consumers and providers do not know each other. The resource allocation is not affected by social relationships but solely depends on the characteristics of the devices and the environment. Resource consumers have no control over the decision on which device their task is executed and, in turn, resource providers cannot decide who runs which task on their machines. These systems require a high level of security as the anonymity might lead to malicious behavior. As there are no social incentives to participate, anonymous systems often provide a compensation for providing resources. This can either be reciprocal as in file sharing systems or monetary.

**Project-based:** In volunteer computing for scientific projects, users can decide to which project they want to contribute [9]. Therefore, the project owners advertise their undertaking and explain how this project can be beneficial for the public. Resource providers then select one or multiple projects. Typically, these projects rely on altruistic resource sharing and motivate contributors by introducing gamification elements like leaderboards. Even though the incentive for malicious behavior is low in these systems, there is no explicit relationship of mutual trust between the volunteers and the project owners. Thus, security mechanisms are required.

**Social Networks:** Friendship relations in social networks are often based on real-world relationships [215]. Resource owners might rather be willing to provide their computational power to their friends and family than to strangers. Thus, the virtual relationships of social networks can be leveraged to match resource consumers and providers. The relationships in the network represent the social closeness between two users. Further, the relations can be labeled or grouped so that users can distinguish between family, close friends, and distant acquaintances. Tasks can then be scheduled accordingly, where confidential tasks will only be executed by very well trusted contacts. Besides leveraging existing social networks, there can also be networks that are designed and implemented for the purpose of resource sharing. Given that there is a certain level of trust between friends in social networks and issues such as the free-rider problem or privacy concerns become less relevant [239, 265].

**Personal Contact:** Tasks can be offloaded in one-to-one relationships where a provider explicitly grants another user access to resources. One possible scenario is sharing between people gathered in one room or in proximity. This solution borrows from the concept of opportunistic computing [266]. The level of trust can be considered very high in these systems.

### 7.1.3. Obstacles

Obstacles are reasons that would discourage users to participate in volunteer computing systems. They are counter players to incentives and social relations when it comes to the question whether users participate in social computing systems or not. We discuss several of these reasons in the following.

**Costs:** Users might simply not want to share resources as the additional workload would require more energy and thus result in higher costs. Especially when there is no compensation, sharing resources is comparable to donating money.

**Security:** Even though many distributed computing systems are sandboxed and are not considered a security threat, many people have mixed feelings about running unknown code on their devices [225].

**Lack of Motivation:** Users might not see any reason why participating in distributed computing systems could be beneficial for them. They might never have experienced situations in which they would have required more computational power and thus do not expect any benefit from a resource sharing system [237]. Further, the amount of money that they could earn by renting out their resources in commercial systems might not be sufficiently incentivizing.

**Effort to Participate:** Sharing resources requires at least some manual setup. Users need to install a software that allows to share resources and register an account [235, 237]. In volunteer computing, users also have to select one or more projects they would like to contribute to. When resources of mobile devices are shared, device owners might need to charge these devices more often which leads to additional effort [225].

**Device Slow-Down:** As each device has a finite amount of processing power, sharing resources might result in a slower execution of user applications. Thus, users might experience a slow-down of their own applications due to the execution of other people's tasks.

### 7.1.4. Categorization of Existing Systems

In this section, we categorize existing social computing systems according to two dimensions: (i) type of incentives and (ii) social relationships involved. We have presented these dimensions in detail above. There are numerous social computing systems available that vary in their purpose, design, application areas, and also in their success. We discuss a subset of all available systems that we consider most relevant for the evolution of social edge computing. Further, we locate each system in the two-dimensional matrix shown in Figure 7.1 and subsequently discuss our findings.

Folding@home is a volunteer computing project for disease research. In 2007, the project was recognized as the most powerful distributed computing network [267] with temporarily more than 400,000 active devices [268]. The website of the project is well maintained and provides reasons why users should donate their resources for medical research. In [268], the authors argue that volunteers want to see something in return and, thus, the results of the projects can be traced in peer-reviewed publications. Besides the altruistic aspect, the project also introduces gamification elements. In hourly updated statistics, the donors can track their own performance and also compete in teams against other teams. To reduce the effort for the users, the Folding@home software is easy to install and does not need to be maintained. Also, clients can manually set preferences for how their resources should be used. This reduces the concerns about running workload on one's own devices.

The *Berkeley Open Infrastructure for Network Computing* (BOINC) is a volunteer computing platform that has achieved similar success as Folding@home [9] with about 800,000 active devices (as of December 2017 [241]). The platform allows to contribute to multiple scientific projects from which SETI@home is the most used one [1]. The platform is easy to use and also provides gamification elements like leader boards for individuals and teams. Anderson *et al.* experienced that users are highly motivated by credits and by comparing their achievements relative to others. To ease the support for multiple projects, extensions like GridRepublic[1] have been added.

Process Thru Processors [269] is a BOINC-based Facebook application by Intel that allows users to contribute their computational resources to selected scientific projects. Users can share their statistics directly in their Facebook news feed which introduces reputation as an incentive mechanism. Very similar to the two projects above is the *Great Internet Mersenne Prime Search* (GIMPS) project that aims at finding new large Mersenne prime numbers [270]. In addition to gamification elements, users are incentivized by a prize money that they receive for discovering a new Mersenne prime number.

In [271], the authors argue that downloading and installing software for volunteer computing would keep users from participating. Thus, they introduce CrowdCL, an open source, web-based volunteer computing framework. However, the system does not include any social relations or incentive mechanisms and has never reached the success of the systems presented above.

In [263], Chard *et al.* introduce the concept of the *Social Cloud* where friends in a social network share their resources among each other. As the users are already friends in this social network, they are likely to have established a certain level of trust and would accept a mutual resource usage. The authors propose to leverage these trust relationships and combine them with further incentive mechanisms such as financial payments or resource bartering. They develop a service marketplace where resources can be traded among friends.

In [234], the idea of the Social Cloud is extended by reciprocal sharing and preference-based resource allocation strategies. In the so-called *Social Compute Cloud*, Facebook users can explicitly or implicitly define the closeness of their friendship relationships. Strong relationships are then prioritized in the resource allocation process. OurGrid [91] implements a network of favors, that is a system of reciprocal resource sharing among peers. Each peer maintains a balance for each known peer in the network and it prioritizes those peers that have provided the most resources.

A famous peer-to-peer protocol is the file sharing technology of BitTorrent [272]. Even though it is not designed for distributed computation it should still be

[1]GridRepublic: www.gridrepublic.org, accessed: 20/03/2019

mentioned here as it is one of the most popular examples for reciprocal P2P file sharing protocols that is robust to the problem of free riding. Teixeira *et al.* introduce a reciprocal-based user provided cloud computing model [273]. Users can share their idle resources to a large pool of cloud resources. In return, they can use resources from this pool. Buttan and Hubaux propose *Nuglets* as a virtual currency to encourage resource owners in ad-hoc networks to collaborate [274]. They implement charging and rewarding mechanisms to account for resource usage. In this reciprocal-based sharing system, users can benefit from the network of devices in their proximity when they contribute to the system themselves. Thus, the deployment of a virtual currency stimulates the participation in the network.

Trust as a virtual currency among strangers is discussed in *Trustos* [262]. Actively participating in a mutual resource sharing system will increase one's own level of trust. Decentralized digital Cryptocurrencies such as *Bitcoins* [275] or *Ether* [276] often require a high computational effort. Resource owners can earn money by mining units of these currencies which can be categorized as public projects. Even though they do not fulfill a scientific purpose such as the projects in BOINC or Folding@home, they work in a similar way. However, their incentive mechanism is purely monetary. POPCORN is an early approach for globally available distributed computation over the Internet [277]. Programmers can request computational power from the POPCORN market where resource owners offer their CPU time in return for a virtual currency called popcoin [278]. The authors claim that this virtual currency could easily be mapped to a real currency. However, this has not been implemented and the system has never left the state of a research project.

*Community Networks* are a form of a decentralized, self-managed infrastructure where users contribute their resources for a common goal. These networks often rely on social ties as they are limited in their size and users need to be in proximity to each other. Kahn *et al.* introduce an incentive mechanism for community clouds where members are not rewarded by the absolute contribution but relative to their capacity [279]. This reciprocal sharing approach also has a social component and establishes fairness within these networks. Ali *et al.* propose a model that enables users to share unused cycles of cloud resources based on trustworthy relationships [280]. They argue that even small virtual cloud instances are idle for some time and that this computational power could be rented. Therefore, they implement a *Cloud Resource Bartering* model that makes use of a virtual

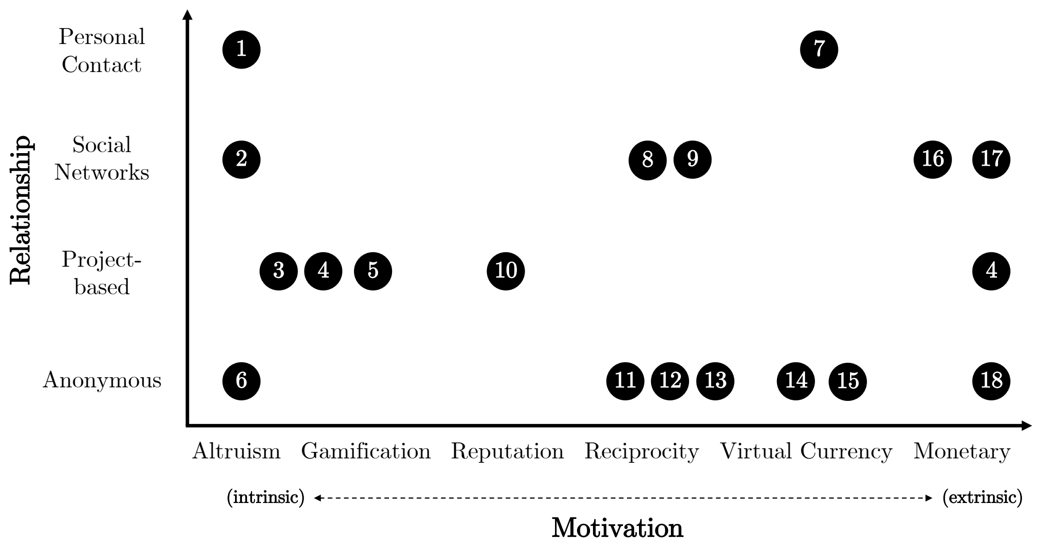


Figure 7.1.: Categorization of resource sharing systems.

| | | | |
|---|---|---|---|
| 1 | SCAMPI [281] | 10 | Progress Thru Processors [269] |
| 2 | Social Volunteer Computing [282] | 11 | OurGrid [91] |
| 3 | Folding@Home [267] | 12 | BitTorrent [272] |
| 4 | GIMPS [270] | 13 | User Provided Clouds [273] |
| 5 | BOINC [9] | 14 | Trustos [262] |
| 6 | CrowdCL [271] | 15 | POPCORN [278] |
| 7 | Nuglets [274] | 16 | Social Cloud [263] |
| 8 | Community Networks [279] | 17 | Cloud Resource Bartering [280] |
| 9 | Social Compute Cloud [234] | 18 | Bitcoin [275], Ether [276] |

currency to trade these resources between users. The authors imply that there is an intrinsic trust relationship between friends in social networks.

In *Social Volunteer Computing*, members of a social network can share their computational resources with each other [282]. The system adopts the characteristics from volunteer computing and adds a social component on top that allows collaborations between friends in the social network. The authors have implemented a Facebook application that uses the existing friendship relations to allocate resources to tasks. This allocation follows a socially-driven approach. The tasks of the resource provider itself are prioritized before tasks of friends or 'friends of friends' are executed. The system also has a social dimension which encourages users to participate as the results of the computations can be directly shared on Facebook.

In SCAMPI (Service platform for social-aware mobile and pervasive computing) [281], Pitkänen *et al.* propose an offloading framework for the immediate environment that takes social relationships and the user's context into account when making offloading decisions. They introduce a *human social layer* that strongly connects the computing resources and the people who own the devices. Based on data from social networks, the system draws conclusions about social relationships between users and takes these connections into account when making offloading decisions.

### 7.1.5. Discussion

Above, we have presented different social edge computing systems and have categorized them according to two dimensions, (i) type of incentive and (ii) social relationships. These categorizations provide some noteworthy insights.

First, even though the term *Edge Computing* had not been established until the year 2014, there were similar systems before. Due to the increase in performance of computational devices, the idea of leveraging resources from user-owned devices still has a big potential. With the continuously growing network infrastructure, edge devices will become easier to access and to include in large computing networks.

Second, only few systems have actually been launched in real-world settings and have remained successful over many years. Whereas most reciprocal-based approaches rather have an academic nature and have not been used on a larger scale, volunteer computing for scientific purposes has been effectively used in ongoing research projects. It might be surprising to see that users obviously participate most in projects, in which they do not receive any monetary reward. However, these volunteer computing systems are well maintained and users can understand that their contribution has a valuable impact. As discussed in [218], this can enhance their intrinsic motivation and lead to rewards like the feeling of a warm glow. Resource owners might not even see the need for a reciprocal resource sharing system as most applications that we run in everyday life would not require more resources than a single device could provide. If, in the future, offloading workload from private devices becomes automized, reciprocal sharing systems might receive more attention in real world systems.

Third, for the longest time, renting out idle CPU cycles has not been profitable enough to become a mass phenomenon. However, with the rise of cryptocurrencies, many resource owners start 'mining' units of these currencies. If the trend of these currencies continues, mining applications can become strong competitors to unpaid scientific volunteer computing projects.

Fourth, as the presented systems are very different in their nature, it is not possible to determine which type of incentive works best in social edge computing systems. To learn more about how potential users see social edge computing systems and to evaluate the potential of such a system, we have conducted an experiment which we present in the following section.

## 7.2. The Effect of Monetary Incentives

The findings from the previous sections show that multiple types of incentives might affect the willingness to contribute to a resource sharing system. To understand the impact of these incentives, we have conducted a survey and have designed and pretested a field experiment which can be found in Appendix B. The findings from these studies show that money is one of the key drivers for participation. However, the general role of monetary incentives for sharing is not yet well understood. Haas et *al.* [283] suggest that different types of social relationships require different incentives for sharing in P2P computing but do not provide empirical evidence. Extant research has revealed that extrinsic rewards can lead to both a crowding-in as well as a crowding-out of the intrinsic motivation to engage in prosocial behaviors such as sharing. In this section, we answer the question whether monetary incentives increase or decrease the resource providers' willingness to participate in peer-to-peer volunteer computing systems. Drawing from relational models theory [284, 285] and motivation crowding theory [286, 287], we derive a conceptual framework in which we propose that the effect of monetary incentives in volunteer computing systems depends on the relationships between resource providers and consumers as well as on a personal predisposition of the resource providers, namely their moral identity centrality [288].

To test our hypotheses we conducted a $2 \times 2 \times 3$ between subjects scenario experiment in which we manipulated whether respondents received a monetary reward or not, whether they shared resources with anonymous users or friends, and whether the system used gamification elements or not [289][2]. Before, we had run a pre-test that allowed us to learn about the general attitude towards computational resource sharing which we will discuss next.

### 7.2.1. Pre-study

In a first step, we decided to conduct a pre-study amongst potential users to validate the results from the student sample in the ex-ante survey in the previous section. We asked respondents about (i) their awareness of possibilities to share computational resources, (ii) their motives and willingness to participate in such sharing systems, and (iii) potential obstacles that would hinder them from participating. We recruited our participants via Amazon's Mechanical Turk [290]. The sample comprises 208 US-based respondents. The mean age in the sample was 37.87 years and 49% of the respondents were female. The respondents received .20$ as compensation for participating in the survey that took approximately 6 minutes on average.

To establish an equal understanding of what sharing of computational resources means, we included an explanatory text at the beginning of the questionnaire. The text outlined the idea of computational resource sharing and briefly summarized the advantages of these systems. First, we asked respondents whether they owned a smartphone, which computational tasks they typically performed on their devices, and whether they feel that limited computational power is a problem for them. Although respondents on average did not agree that limited computational resources posed a problem for them, they agreed that they had experienced that their smartphone's battery drained faster due to a computationally intensive application and felt that an application might be faster if their phone had more computational resources. Further, to ensure that all respondents understood that sharing can work in both directions, that is, respondents can use other users' computational resources but can just as well be providers of computational resources, we included an explanatory text.

Next, we asked respondents whether they were aware of the possibility to share computational resources and whether they had made past experiences with sharing.

[2] [289] is joint work with L. M. Edinger-Schons, D. Schäfer, A. Stelmaszczyk and C. Becker

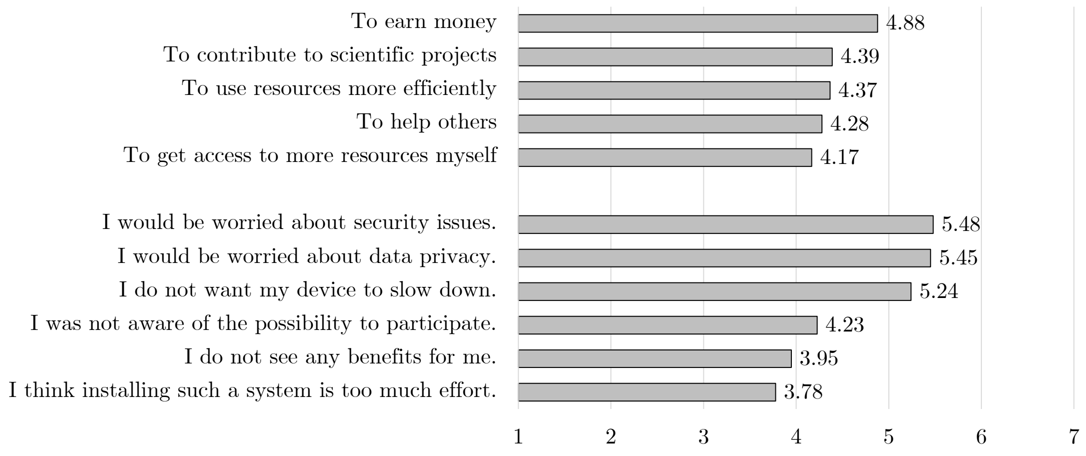


Figure 7.2.: Reasons (top) and obstacles (bottom) to share computational resources.

50% of the respondents reported to be aware of the possibility to share computational resources with others and 26.9% of these respondents had made experiences with sharing of computational resources. Asked which kind of experience they had made, most of these respondents reported to have participated in SETI@home and 80% explained that the experience with sharing of computational resources had been a positive one. Moreover, we were interested in the respondents' future willingness to share computational resources with others. Specifically, we asked them about their general willingness to share computational resources with other users, their willingness to share depending on social relationships and incentives, and potential obstacles to sharing.

We further directly asked respondents what their motivation would be to participate in a system to share computational resources. Here, results reveal the highest agreement for *'to earn money'*. The more altruistic motives *'to contribute to scientific projects'*, *'to use resources more efficiently'*, and *'to help others'* rank second, third, and fourth, while *'to get access to more resources myself'* received the least agreement (compare Figure 7.2 (top)).

We were also interested to know which obstacles would prevent respondents from participating in sharing resources with other users. In a descriptive analysis of the data, the strongest agreement can be found for security issues, data privacy, and worries that one's own device could slow down (compare Figure 7.2 (bottom)).

In terms of social relationships, we were interested to know whether respondents felt that varying compensation mechanisms would be appropriate for different

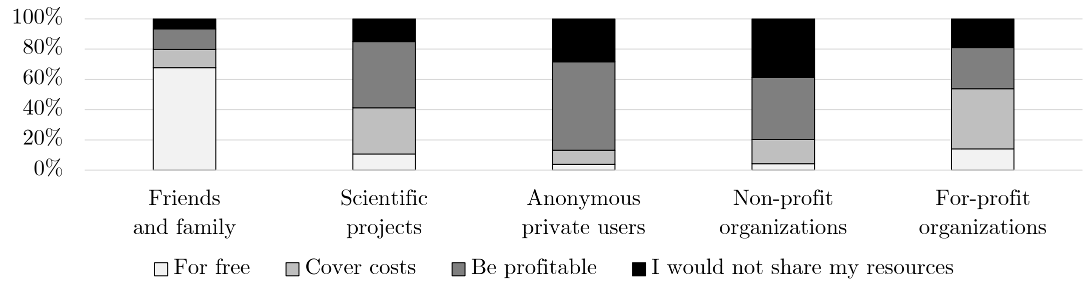


Figure 7.3.: *Which compensation would you demand from the following users for sharing your resources?* For each of the four user groups participants could decide whether they would share their resources for any kind of compensation or whether they would not share them at all.

groups of users. To introduce the possibility of varying compensation mechanisms, we explained to them: *'Imagine a system that allows you to safely share your computational resources with others. Now, imagine a system in which device owners can be compensated for sharing their resources. They can be compensated in multiple ways: i) They could share their resources for free, which means that they do not receive a compensation. ii) They could receive a compensation to cover their energy costs. iii) They could receive a compensation to make a profit. Which compensation would you demand from the following users for sharing your resources?'*

With friends and family, 67.8% of respondents would share computational resources for free. Interestingly, in case of scientific projects, for-profit organizations, and anonymous private users, the majority of respondents (43.7%, 58.3%, and 41.1%) would want the exchange to be profitable for them. In case of non-profit organizations, the majority (39.8%) would just expect their costs to be covered. Further, in case of scientific projects and non-profit organizations, a considerable share of respondents would be willing to share their computational resources for free (10.7% and 14.1%) whereas this is not the case for for-profit organizations and anonymous private users (compare Figure 7.3).

These results are mostly in line with the outcomes of the student-based survey in Appendix B. The only major difference is that respondents from the MTurk sample would rather share their resources for a small compensation with for-profit organizations than with non-profit organizations or scientific projects. In the student sample, this was the other way round. One possible reason could be the students' proximity to an academic institution.

### 7.2.2. Empirical Study

The pre-study revealed some interesting insights which motivated our following work in which we focused on P2P systems. Interestingly, monetary incentives seem to play an important role, given that respondents rate *earning money* as the most important motivation to engage in resource sharing. However, the type of social relationships can obviously not be neglected as it seems to determine the compensation that users expect from their peers. In the following, we will derive a framework of the effectiveness of monetary incentives in P2P computational resource sharing networks and test it in a between subjects experiment.

#### Hypotheses development

Extrinsic incentives have been found to be effective to enhance prosocial behaviors such as resource sharing [218,291]. However, there is a considerable large literature stream which has revealed that extrinsic incentives may lead to both, a crowding-in or a crowding-out of the intrinsic motivation to engage in prosocial behaviors [287, 292]. In other words, monetary rewards can strengthen or harm the intrinsic motivation to do good deeds.

We propose that the question whether monetary incentives enhance or reduce the willingness to participate in volunteer computing depends on the type of relationship between resource providers and consumers. We draw from the *relational models theory* [284, 285] to make this argument. Heyman and Ariely (2004) explain that humans categorize interactions as either social or monetary and that their reactions to monetary incentives depend on this categorization. If the relationship is categorized as social, providing a monetary incentive may harm accepted norms of behavior [285]. To illustrate, imagine the following situation: If your best friend asks you to help her with her house move, you would most probably not consider to charge her an hourly fee.

In line with this literature, we expect monetary incentives to increase the willingness to participate in volunteer computing if the consumer group is made up of anonymous users. This is due to the fact that the exchange will be categorized as a monetary transaction and monetary incentives will be considered appropriate. In contrast, we expect monetary incentives to reduce the willingness to participate in volunteer computing if sharing takes place amongst friends. In this case, the exchange resembles a social interaction and receiving a monetary reward would harm social norms of friendship namely that you should not benefit from your friends financially. Thus, in summary, we expect monetary rewards to lead to a crowding-in of the motivation to participate for anonymous private users and to a crowding-out for sharing amongst friends.

Finally, over and above the interaction between compensation and relationship type, we argue that these effects depend on the personal predisposition of the individual user. Specifically, we expect the users' centrality of moral identity [288] to be a decisive factor. The centrality of moral identity is conceptualized as "*the cognitive schema a person holds about his or her moral character*" [p.124] [293]. It is stored in a person's memory as a complex knowledge structure consisting of moral values, goals, traits, and behavioral scripts [288, 294]. These knowledge structures are assumed to be acquired through life experiences and therefore to differ across individuals [295]. For people whose moral identity occupies greater centrality within the self-concept, being a moral person is more self-definitional compared to other identities [296].

We expect that individual differences in the centrality of a person's moral identity will play a key role in determining their reactions to monetary incentives in the context of volunteer computing. Individuals with a high centrality of moral identity will strive for consistency with their ideal of being a moral person. This means that they will try to adhere to accepted social norms of behavior. For these individuals, a monetary incentive for sharing resources with friends will likely lead to a reduction in the willingness to participate (i.e., crowding-out). In a monetary interaction, however, the monetary incentive will strengthen these users' intention to participate (i.e., crowding-in). Individuals with a low centrality of moral identity on the other hand, will put less value on following such rules of social interactions. For these individuals, a monetary incentive will most probably have a positive effect and strengthen their willingness to participate regardless of the type of relationship.

To summarize, we expect that in case of anonymous private users, monetary incentives will lead to a higher motivation to share for participants with a high moral identity than for those with a low moral identity. Conversely, we argue that that amongst friends, monetary incentives will lead to a lower motivation to share for participants with a high moral identity compared to those with a low moral identity. We formally hypothesize:

**Hypothesis 1 ($H_1$):** Monetary incentives enhance the willingness to participate in volunteer computing.

**Hypothesis 2 ($H_2$):** The type of relationship (i.e., anonymous users versus friends) moderates the effect of monetary incentives in a way that monetary incentives have a positive/negative effect on the willingness to share computational resources with anonymous users/friends.

**Hypothesis 3 ($H_3$):** We propose a three way interaction between monetary incentive, type of relationship, and moral identity. More specifically, we hypothesize that in case of anonymous users/friends, monetary incentives will lead to a higher/lower motivation to share for participants with a high moral identity than for those with a low moral identity.

### Scenario Experiment

We decided to test our framework in an online scenario experiment which we will report next. In this study, we asked participants to imagine a system in which they could share computational resources with other users. We experimentally manipulated whether they would receive a monetary reward or not and whether they would share their resources with anonymous private users or with their friends. We measured individual differences in the centrality of moral identity using an established psychometric scale. We also included the questions from the preliminary study to replicate the findings from the descriptive analysis with a larger sample (results are consistent but we do not report them for the sake of conciseness). Further, we decided to manipulate elements of gamification as this is a highly relevant question for the designers of such systems. Making the users' own as well as the contributions of others transparent may enhance the users' motivation and engagement due to gamification [244].

**Design:** We thus conducted a $2 \times 2 \times 3$ between subjects scenario experiment (i.e., no monetary compensation versus monetary compensation; sharing with anonymous users versus with friends; no gamification; statistics; leaderboard). The design is illustrated in Figure 7.4.

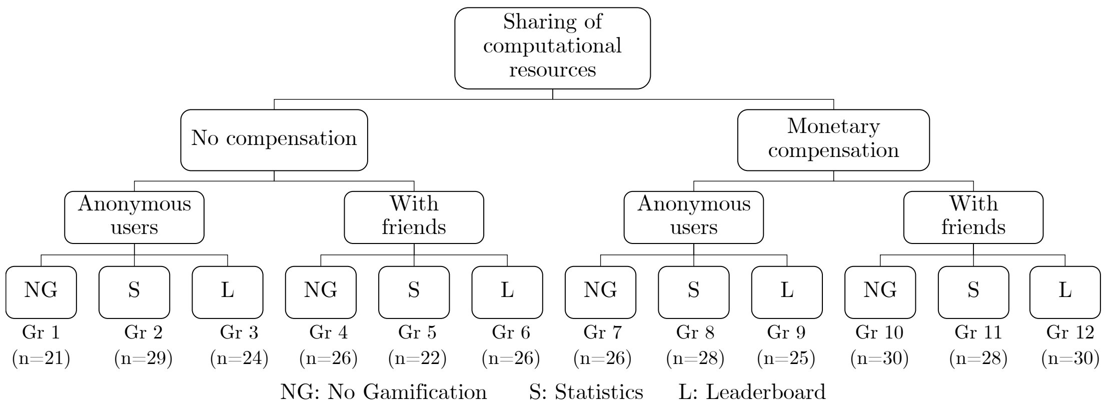


Figure 7.4.: Design of the 2 × 2 × 3 between subjects experiment. (NG: No Gamification, S: Statistics, L: Leaderboard)

**Procedure:** We conducted the study as an online survey, programmed as a series of webpages. The invitation to participate in the study was sent out via Amazon's MTurk. Participants first read the scenario texts and then rated a set of Likert-type items (all ranging from 1 to 7). We measured participants' willingness to participate in such a system, manipulation checks, the questions from the pre-study, and moral identity (in this order). Finally, we elicited demographic information.

**Treatment materials:** We used the following treatment materials. All groups received an introduction which reads as follows: '*Imagine a system that allows you to safely share computational resources.*'. As a manipulation for relationship type we added: '*Via this software package you can share computational resources with 1) friends; 2) anonymous users.*'. To manipulate the compensation we further explained: '*The sharing of resources is based on a system of 1) monetary compensation, i.e., you receive money for the computational resources which you share with other users in the system and you pay for the resources that you use or 2) reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free.*' Last, as a manipulation of the gamification element, we included three levels. In the 'no gamification' condition, we did not add any further information. In the 'statistics' condition we explained: '*The software is further designed to transparently track your provision of resources. On a statistics page you can continuously track the amount of resources that you provided to other users and the amount of resources of others that you have used.*' In the 'leaderboard' condition the text reads as follows: '*The software is further* *designed as a game in which you compete with other users for the highest score. You can increase your score by providing resources to others. On a leaderboard you can see your performance compared to those of other users. Each day the best-performing contributor receives a symbolic award and earns extra points for the leaderboard.*' The full experimental material can be found in Appendix A.

**Measurement:** To capture participants' willingness to participate, we included one straightforward item: 'How likely would you be to participate in the system?' with scale anchors ranging from '1=*not likely at all*' to ''7=*very likely*'. To measure respondents' centrality of moral identity, we used four items from an established scale developed and tested by Aquino and Reed [288]. Participants were first provided with the following instructions: '*Listed below are some characteristics that might describe a person: Caring, compassionate, fair, friendly, generous, helpful, hardworking, honest, kind. The person with these characteristics could be you or it could be someone else. For a moment, visualize in your mind the kind of person who has these characteristics. Imagine how that person would think, feel, and act. When you have a clear image of what this person would be like, please report in how far you would agree with the following statements.*' Then, they were asked to rate their agreement with the following items: *1. 'It would make me feel good to be a person who has these characteristics.' 2. 'Being a person who has these characteristics is an important part of who I am.' 3. 'I am actively involved in activities that communicate to others that I have these characteristics.' 4. 'I strongly desire to have these characteristics.'* The items were all rated on a seven point Likert scale ranging from 1 ('*I do not agree at all*') to 7 ('*I fully agree*'). The scale has a good internal consistency with a Cronbach's alpha value of .862 and all factor loading are above .649.

**Sample:** 498 US participants took part. The length of the survey was estimated to be approximately 15 minutes. In the MTurk sample, on average, it took participants only 10 minutes to answer the full questionnaire. We excluded those participants who answered the questionnaire in less than 8 minutes (183 individuals) because it is unlikely that they answered the questions attentively. As a consequence, we had a final sample of 315 valid responses. The mean age is 36.78 years and 47.9% of the respondents were female. The following table summarizes the exact sample sizes in all cells:

**Manipulation checks:** We included manipulation check items to make sure that our manipulations worked as intended. We used ANOVA to test for differences across groups. Using a dummy-coded variable (0 for anonymous users and 1 for friends) as an independent and the first item as a dependent variable we found a significant difference ($m_{anonymoususers} = 4.66$; $m_{friends} = 5.25$; F $(1, 315)$ = 7.837, p=.005). The same applies for the second item ($m_{anonymoususers} = 4.80$; $m_{friends} = 3.54$; F (1, 315) = 29.174, p=.000). To check whether respondents understood the monetary compensation manipulation, we evaluated whether respondents expected a monetary compensation in return for the resource sharing or whether they considered sharing their resources for free. Using a dummy-coded variable (0 for no monetary compensation and 1 for monetary compensation) as an independent and the first manipulation check item as a dependent variable we found a significant difference ($m_{nocompensation} = 3.35$; $m_{monetarycompensation} = 5.40$; F $(1, 315) = 89.549, p = .000$). This is also the case for the second item ($m_{nocompensation} = 4.95; m_{monetarycompensation} = 3.13; F(1, 315) = 62.243, p = .000$). Finally, we checked whether respondents understood the different types of gamification elements. For the leaderboard item, we observed ($m_{nogamification} = 2.93$; $m_{statistics} = 3.03$; $m_{leaderboard} = 5.45; F(2, 315) = 55.091, p = .000$). The results of the statistics page were ($m_{nogamification} = 4.70$; $m_{statistics} = 4.81$; $m_{leaderboard} = 3.48; F(2, 315) = 15.120, p = .000$). Thus, the manipulation checks indicate that all manipulations worked as intended.

**Analysis:** To analyze our data, we first used ANOVA to check whether our treatments had significant effects on the central dependent variable, i.e., the willingness to participate in volunteer computing. In a second step, we computed moderated regression analyses using SPSS Process, model 3, to test for the hypothesized three-way interaction.

**ANOVA results:** In a first step, we ran ANOVAs to test for the effect of the treatments on willingness to participate. Supporting $H_1$, the monetary incentive significantly enhances willingness to participate (F $(1, 315) = 7.844, p = .005$). Further, in line with $H_2$, we found a significant two-way interaction between monetary incentive and type of relationship (F (1, 315) = 3.985, p=.045). Including a binary variable for moral identity centrality (MIC) (0= low MIC, 1= high MIC, based on a median split, median =5.5), we found a significant three way interaction between monetary incentive, type of relationship, and MIC (F

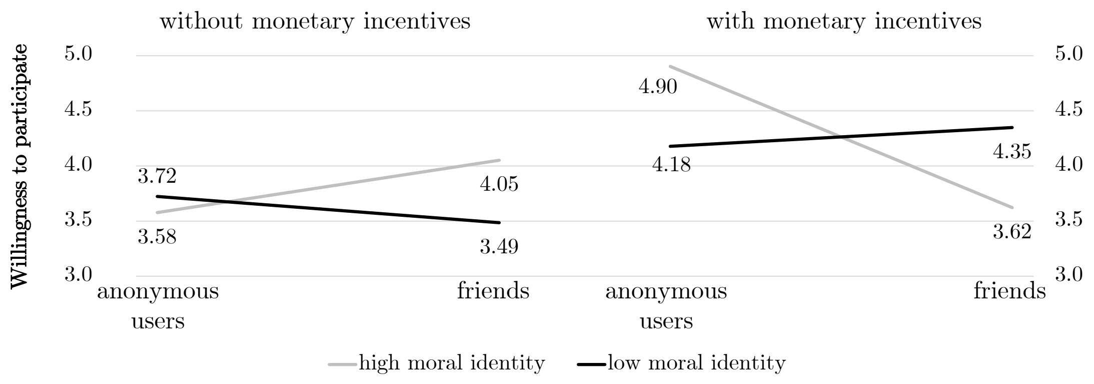


Figure 7.5.: Willingness to participate in volunteer computing across groups.

$(1, 315) = 5.950, p = .015)$, providing first evidence for $H_3$. Figure 7.5 presents the means of these groups.

Concerning the manipulation of gamification elements, we neither found direct nor interactive effects with the other treatments. Thus, we merely controlled for the manipulation in the analyses.

**Moderated Regression Results:** To formally test for the moderating role of moral identity centrality, we implemented a moderated regression analysis in SPSS process, model 3, with Y=willingness to participate, X=monetary incentive, W=relationship type friends, and Z=MIC. The results of the model estimation fully support our theorizing. The two-way interaction ($b = 4.35, p = .022$) as well as the three way interaction ($b = -.93, p = .005$) are significant. Further, simple slopes analyses reveal interesting insight into the effect of the independent variable monetary incentive on the dependent variable of willingness to participate for different levels of the moderators.

Figure 7.6 graphically illustrates these results. In case of anonymous users, the effect of monetary incentives is insignificant for lower levels of MIC and significantly positive for mean and higher levels. For the case of sharing with friends, in contrast, the effect of monetary incentives is significantly positive for lower levels of MIC, insignificant for mean levels, and significantly negative for higher levels. Whereas the coefficients of the effect of monetary incentives on willingness to participate for anonymous private users are not significantly different from each other, the coefficients for friends do differ significantly indicated by confidence intervals which do not overlap ($b_{lowMIC} = 1.07, [.3102; 1.8235]; b_{highMIC} = .086, [-1.6906; -.0282]$.

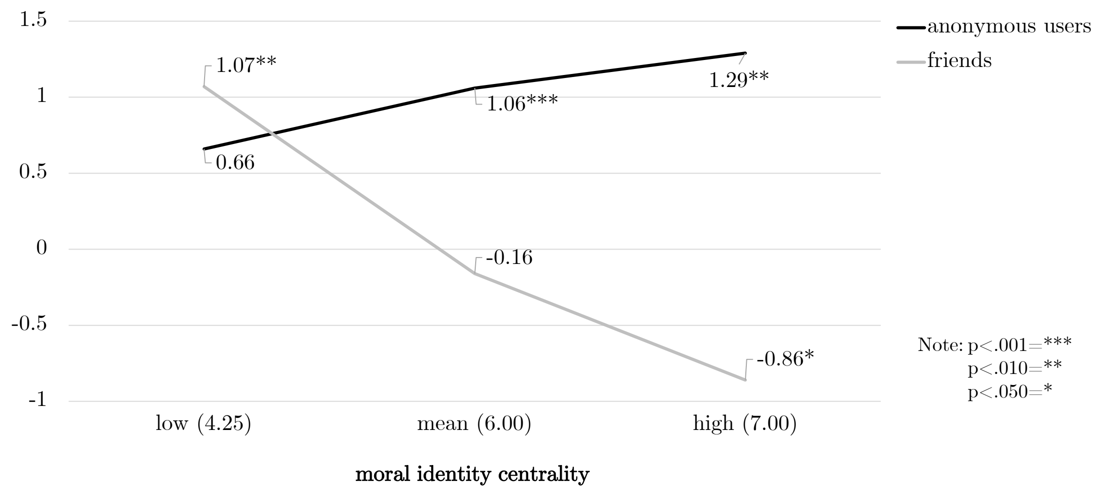


Figure 7.6.: Simple slopes: the effect of monetary incentives on willingness to participate for different social relationships and varying levels of moral identity centrality.

### 7.2.3. Discussion

Our results have implications for academic theory and for the design of platforms for computational resource sharing alike. To the best of our knowledge, this experiment is the first which combines the two theories of relational models and motivation crowding to explain sharing behavior. While P2P computing is only one application area among multiple examples, further research is necessary to validate these results in other parts of the sharing economy where sharing happens among strangers and friends.

In terms of conceptual contributions, our results contribute to four literature streams. First, we contribute to the literature on P2P platforms in the sharing economy. The results of our scenario experiment indicate that monetary incentives may have detrimental effects on participation of the type of relationship between users is a social relationship and thereby tie in very well with recent findings on social versus market-based exchanges in the sharing economy. Second, our study links to the evolving stream of research on relational models in general by providing another application in which these models play a decisive role [297]. Third, only very few papers have so far scrutinized the factors that moderate the effect of extrinsic incentives on prosocial behaviors. While conceptual work has laid out the mechanisms by which crowding-in and crowding-out effects occur, only very little empirical research has so far identified and empirically tested moderating variables that explain this 'flip' (e.g., [298]). Fourth, by testing moral identity centrality as a moderator in our model, we contribute to the incipient research stream that proposes this construct to be an important determinant of prosocial behaviors [293].

In terms of practical implications, our results suggest that software developers who intend to design tools for the sharing of computational resources should take into account that monetary incentives can enhance or harm willingness to participate depending on social relationships and personal characteristics. Thus, such a system needs to provide the possibility for participants to select whom they want to share their resources with and what compensation they expect in return.

As every study, this study has limitations which at the same time represent avenues for future research. First, our scenario experiment relies on an intentional measure as dependent variable. Intentions and behaviors may differ and it would be a necessary next step to replicate this study's results in a setting that includes objectively measured outcomes. This can be done in a field-experiment with a real application resource sharing. This would also allow for the analysis of more diverse behavioral dependent variables. We expect gamification to show an effect when applied in a field study. As gamification leverages peoples' emotions and desires it can hardly be simulated in scenario experiments but only reveals its full potential in real world settings. We have designed such an experiment an ran a pretest with a small sample (compare Appendix B). Second, we only tested for the effect of monetary rewards versus no monetary rewards while the spectrum of possible incentives is much broader. A follow-up study could test various incentives ranging from more intrinsic to more extrinsic rewards and observe the respective motivation crowding-effects. Finally, while we chose to focus on moral identity centrality, there could be other important personality factors that we neglected. Future research could set out to test other possible individual-level factors that capture unexplained variance.

## 7.3. Social Computation Sharing Platform

In the previous section, we conducted an experiment to understand the role of incentives and social relationships in computational resource sharing systems. We analyzed existing sharing systems in terms of the incentive mechanisms and the relationships between resource consumers and providers. We used a scenario experiment to find out about the effect of monetary incentives on the willingness to share resources with anonymous users and friends. The results leave no doubt that incentives and social relationships play an important role in the decision whether private device owners participate in computational resource sharing. The results are in line with previous work that suggests that social relationships can be leveraged to infer a level of trust between users and, thus, to increase participation in resource sharing [207, 215]. The findings from the scenario experiment also support the proposition of Haas *et al.* that users expect different kinds of compensations from resource consumers in different kinds of social relationships [283].

We transferred these results to the Tasklet system that did not include any of these aspects but was purely concerned about technical aspects of resource sharing. However, in edge computing, where private device owners act as resource providers, incentives and social relationships cannot be left aside anymore. Thus, in this section, we propose and implement an accounting system and a social network as an overlay on top of the Tasklet system. The accounting system introduces market mechanisms into the resource sharing system, monitors the exchange of computational resources, and enforces the compensation in return for the resources. In the social network, users can establish friendship relationships which allows a scheduler to infer a certain level of trust and to charge different prices for resource consumers depending on the closeness of the users' relationship. In combination, the accounting system and the social network turn the Tasklet system into an economic and social resource sharing system.

In this section, we first introduce the friendship model for the social network. We define the nature of friendship relations in the network and discuss how these relations can be used for task scheduling. Afterwards, we discuss our accounting model for the Tasklet system. This includes the questions what kind of compensation is used, what the costs for a Tasklet execution are, how transactions are performed, and, finally, who the responsible entity to perform and monitor these transactions is. Finally, we discuss our system design, including the entities of the accounting system as well as the friendship network, the data management, and the development of the frontend for the social network.

### 7.3.1. Friendship Model

When individuals exchange goods, trust is a central aspect and facilitator to participate in these activities [299]. In real life, individuals have more trust towards closer friends than strangers [300]. Further, friendship relationships in online social networks are at least partly based on social relationships in the real world [215]. As a result, we can assume that the level of trust is higher between users that are connected in these networks than between users that are not related through a virtual friendship. Given the results from our experiments, this means that the willingness to share resources with online friends is higher than with anonymous users. Further, users might not want to charge online friends for using their computational resources. Here, we discuss the characteristics of our friendship model in the Tasklet social network. We define what a friendship relation is and how information about these relationships can be used for Tasklet scheduling and accounting.

#### Dimensions of Friendship Relationships

Multiplexity describes the strength of a relationship. Relationships can be either homogeneous when all friendship relations are considered equally strong or weighted [301]. For weighted relationships, some relations are defined as more important or closer than others. The weight can be determined by different factors. These factors can be either manually defined by the user or automatically retrieved based on similarity or interaction patterns. Similarity-based approaches use the concept of homophily that describes the tendency to establish closer relationships with people we have most in common with [302]. This includes race and ethnicity, as well as age, religion, education, occupation, and gender. In a social network, these sociodemographic dimensions could be leveraged to infer relationships between homogeneous users [303, 304]. Interaction-based models consider the number of messages, wall posts, tags in the same picture, comments, and likes [207, 301, 305–307].

Reciprocity describes whether friendship relations are bidirectional [308]. In a symmetric model, a relationship can only be established when both users agree. Facebook is a prominent example for this model. Relationships in an asymmetric model can be established in a unidirectional way and do not need the confirmation from the other user. The follower principle in Twitter works in such an asymmetrical way. Computational resource sharing might work in both ways. In each relationship, a user can either be the resource consumer, resource provider, or both. A reciprocal system would always assume that consumers are providers as well.

The concept of transitivity in a population goes back to Anatol Rapaport [309]. Transitivity is often described as "*friends of my friends are my friends*" [308, p.133]. In a transitive friendship model, the social relationships of a user's friend are considered to have a certain relevance. In the context of computational resource sharing, users might be more willing to share resources with transitive friends than with anonymous users [310]. Non-transitive models do not consider these friends-of-friends relationships as relevant. In [207], for example, the authors limit the computation to direct friends to increase the level of trust.

### A Friendship Model for the Tasklet Social Network

To build a friendship model for the Tasklet social network, we needed to make a decision for each of the three dimensions multiplexity, reciprocity, and transitivity. The friendship relations have three effects on the system. First, they determine which user is allowed to use the resources of any other user. Second, they infer a certain level of trust which is relevant for scheduling confidential or critical tasks. Third, they might have an impact on the compensation that resource providers request in return for sharing their resources.

Regarding multiplexity, friendship relations are either weighted or unweighted. In case of weighted relationships, resource providers could define a threshold and allow only those consumers with a weight greater than the threshold to use the resources. The weight can also be used to determine the level of trust between the users. Confidential tasks might only be executed on providers with a high level of trust. Further, weighted relations would allow to compute the compensation by the closeness of the users. Thus, weighted relations provide fine-granular sharing settings for resource providers. On the downside, the weights have to either be manually adjusted or inferred by homophily metrics. Manual adjustments introduce additional workload for the users, especially since the relationship strength might vary over time and need to be constantly updated. An automated weighting based on similarity measures might lead to results that contradict the user's opinion about closeness. On top, unweighted friendship relations allow to distinguish between known and anonymous resource providers which might already be sufficient for most users. Thus, we argue that weighted relationships introduce more complexity than they would benefit the system and thus decided to implement binary, unweighted relations.

We further considered the reciprocity of friendship relationships, that is, whether the relations are symmetric or asymmetric. For the Tasklet social network this means that trust could be expressed in a unidirectional way. Users might create a friendship relationship without the assent from the other user. This other user might then consider the first user as target for computation offloading. We argue, however, that resource sharing requires trust from both sides. This reasoning is in line with resource sharing platforms in other domains such as AirBnB and Ebay where private users deal with each other. These systems introduce a bidirectional level of trust by providing public rating for both, the resource provider (seller) and the resource consumer (buyer). Therefore, we decided to implement symmetric friendship relationships in the Tasklet social network.

Finally, social networks can either support transitivity or not. The benefit of using transitivity in the Tasklet social network would be that different levels of trust could be applied without the need to manually define relationship weights. The strength of a relationship between to users could be computed by the shortest path between them. These friends-of-friends relations could serve as a coarse measure of social proximity and trust between the users. Resource owners could set different levels of compensation for direct friends, friends of friends, and so on. They also had the chance to restrict the level of sharing resources to, for example, direct friends. We suggest that transitive relations allow for an automated calculation of social proximity without introducing the complexity of weighted friendship relations. In summary, we designed our friendship model for the Tasklet social network as unweighted, symmetric, and transitive relationships.

### 7.3.2. Accounting Model

Our previous studies have shown that introducing the idea of a compensation can increase the willingness to share computational resources in a peer-to-peer system. While this effect is only minimal for sharing among family members and friends, resource owners are highly reluctant to share resources without getting anything in return (compare Figures B.6, 7.3, and 7.6). Further, evaluations on the file-sharing system Gnutella show that free riding is an important issue when resources are shared for free. In the system that was built on fairness and trust, about 70% of users did not share any data and nearly 50% of resource requests were responded by 1% of the participants [311]. The situation even intensified several years later when the study was repeated and revealed that about 85% of users did not share any files [312]. Thus, overcoming the free riding problem is considered highly relevant for peer-to-peer systems [265].

We introduce an accounting model that allows to compensate resource providers for sharing their computational power and tackles the free rider problem. In combination with the Tasklet social network, the accounting model solves the problem of free-riding. As resource consumers have to compensate providers, the threat of exploiting resource providers by anonymous consumers is eliminated. The social network allows to share resources for free within a group of family members or friends where free-riding is less of a problem. In the following, we present our accounting model.

#### Types of Compensation

We discussed different types of compensations in Section 7.1 including credits for computation, a virtual currency, and a real currency. For the Tasklet system, computational credits could be implemented to allow for reciprocal (i.e., *tit-for-tat*) sharing. As users could be resource providers at one time and resource consumers at another time, computational power could be treated as a virtual good similar to vehicles in a carsharing system [256, 257]. However, since computation was the only accepted currency in such as system, users would be required to provide as many resources as they use from others. At the same time, resource providers with powerful machines who do not require additional resources would accumulate credits that turn out to be useless for them.

In contrast, monetary compensation would be the most flexible approach where users that require more computation power than they provide can pay for additional resources. Resource owners can earn money for sharing their computation power and use this money for any other goods. One drawback of monetary payments are the transaction costs [313]. When using the online payment service PayPal, each transaction is charged with a fixed fee \$0.05 cents plus a variable fee of 5% of the transaction amount [314]. As the price for an hour of computation typically does not exceed few US cents, most transactions in the Tasklet system will be in the range of \$0.001 to \$0.05. The overhead and fees for these transactions would exceed their actual value.

Virtual currencies represent a compromise between reciprocal sharing and monetary compensations. A virtual currency can be mapped to a real currency and can be traded for actual money [274, 313]. For all transactions within the system the virtual currency is used. There are no transaction fees and the monetary value of each transaction might be in the range of less than one cent. Virtual currencies can be found in different domains. Popcorn is an early implementation of a computational resource sharing system for Java applications [277]. Resource providers (*sellers*) offer their resources to anonymous resource providers (*buyers*) who pay for the resources in form of virtual credits called *popcoins* [278]. The authors argue that the virtual currency can be used for different market mechanisms such as donations, trades, barters, loans, and conversions into other currencies. Buttyán *et al.* use virtual currencies, called *nuglets*, to stimulate cooperation in self-organized mobile ad hoc networks [274]. Participants get paid in nuglets for forwarding packets within the network. Vishnumurthy *et al.* present *KARMA*, an economic framework for any kind of peer-to-peer resource sharing [315]. Their main goal is to keep track of resource consumption and resource contribution in order to avoid free-riding. The current balance of each participant is represented by a single scalar value. Hausheer *et al.* use a token-based accounting system that avoids double spending [316]. Resource consumers and providers exchange these tokens that can only be used once in return for any kind of shared resources. The system is demonstrated on the example of a file sharing system. Finally, *Linden Dollar* is the virtual currency in the online virtual world Second Life [313]. Participants in this online environment, called *residents*, can use this currency to buy virtual objects or services from other peers or services from the operator of the environment. Residents can also earn Linden Dollars by selling virtual products or services. Even though this application of a virtual currency is independent from computational resource sharing, it can easily be applied to other domains. It is of particular interest as it has been successfully used for multiple years and several millions accounts were registered at peak times [317].

For the Tasklet system, we decided to use virtual currencies as they make the contribution to the system independent from the consumption of resources and are not subject to transaction fees.

### Market Models

Similar to real world currencies, virtual currencies underlie several similar market mechanisms. However, in contrast to the former, virtual currencies are not affected by external influences or global economic trends. Operators of a resource sharing system have full control over the virtual currency and can enforce their own rules and market mechanisms. For the Tasklet system, we also created a market model which is discussed in the following.

There are several options for participants in a resource sharing system to acquire units of this virtual currency, which we will henceforth refer to as *credits.* In [252], Bogliolo *et al.* suggest that new participants retrieve an initial amount of credits when they join the system. The amount of credits then scales with the number of participants. Credits cannot be purchased and cannot be turned into a real currency. However, to make the system a part of the Internet value chain, the authors introduce the process of *monetization* that allows to trade in the credits for real money. To avoid speculation, credits that are bought with real money can only be monetized after the first transaction. Monetized credits lose their value in the sharing system. *KARMA* uses a similar approach [315], in which participants receive an initial amount of credits. However, KARMA is a closed system and it is not possible to monetize the credits. In the *nuglets* approach, credits can only be earned by sharing resources [274]. However, as credits can get lost over time, the authors suggest the possibility to buy credits for real money. To limit the amount of total credits in the system, the exchange rate can be adjusted accordingly. *Linden Dollars* can be either earned during the game or purchased directly from the system operators [313]. There is no direct exchange mechanism into real currency. Third parties such as PayPal, however, provide ways to monetize the surplus credits [317].

In the Tasklet system, we follow the latter approach but allow for a direct monetization of the credits. This allows resource consumers to purchase computational power at a market price and resource providers get compensated for the additional energy cost which they also have to settle with real money.

In [264], Buyya *et al.* discuss multiple economic models for resource management and a comprehensive categorization of existing systems. In the auction model, an auctioneer manages the negotiation between resource consumers and resource providers. Auctions can be either single-sided where the provider announces a service and invites bids until no consumer is willing to pay a higher price. In double-sided auctions, providers and consumers can submit bids at any time [278]. When there is a match, the transaction is executed. In *Spawn*, a sealed-bid, second-price auction is implemented [261]. Consumers cannot see the bids from other consumers and the winner only has to pay the amount offered by the second highest bidder [318].

In commodity markets, a third party sets the price and consumers buy the resources from a pool of equivalent choices [319]. In bargaining models, consumers directly negotiate with providers for lower prices. In a tender/contract-net model, consumers announce their requirements and invite bids. They take the most appropriate offer and directly contacts the provider [320].

In distributed systems, double spending is one of the most common frauds [252]. To avoid these attacks, the system requires a trusted entity, which can either be decentralized [321] or represented by a trusted third party [278, 322]. Blockchains provide a fully decentralized solution for the accounting resource exchanges [275]. Hybrid approaches use super peers or multiple trusted entities to provide robustness. In [316], a group of trusted super peers signs and validates tokens. In [315], the transactions are performed via trusted banks that are assigned to each participant.

In the Tasklet system, we use a tender/contract-net model which gets mediated by a trusted, centralized broker. As the broker in the Tasklet system has global knowledge about the providers and performs QoC-aware Tasklet scheduling, it can take the price as another non-functional property into account when making allocation decisions. In case of a decentralized scheduling approach (compare Section 6.3) this tender/contract-net needs to be decentralized as well. Resource providers can set their prices individually. They can decide whether friends in the Tasklet social network receive a discount or even use the resources for free. Further, they can also give discounts for transitive friend relationships and exclude anonymous users.

There are two main different approaches to account for task executions. Prices can either be charged by operation or task [278] and are, thus, independent from the required execution time. In contrast, a task execution can also be accounted by the time a CPU is busy [261]. Finally, it needs to be defined what happens in case of a failed transaction. This might happen when the provider stops the execution at any time or when the consumer leaves the system during execution. Bogliolo *et al.* suggest a system that reduces the public trustworthiness in case of a misbehaviour [252]. Another option is to rollback the transaction [315, 316]. However, it might happen that an execution is already performed but the consumer has left the system and would not pay for the execution. In these cases, the provider has already spent energy for the computation without getting compensated.

In the Tasklet system, we follow the approach of Regev and Nisan and charge the consumer for each operation by the Tasklet virtual machine [278]. Failed transactions are rolled back and the resource provider does not get compensated. We acknowledge that the latter is not optimal as the resource performed computation without getting any compensation in return. However, solving this issue is part of future work.

### 7.3.3. System Architecture

The Tasklet social network has two purposes. First, it connects the computational device with actual persons and allows for an accounting of computational resource sharing. Persons have user accounts and can trade the computational power of their devices against credits of a virtual currency. The Tasklet social network performs the orchestration of this exchange. Second, it allows its users to create friendship relationships among each other which can then be used to improve the Tasklet scheduling decisions. Friendship relations can be leveraged to infer a level of trust and send confidential Tasklets only to direct friends. Tasklets

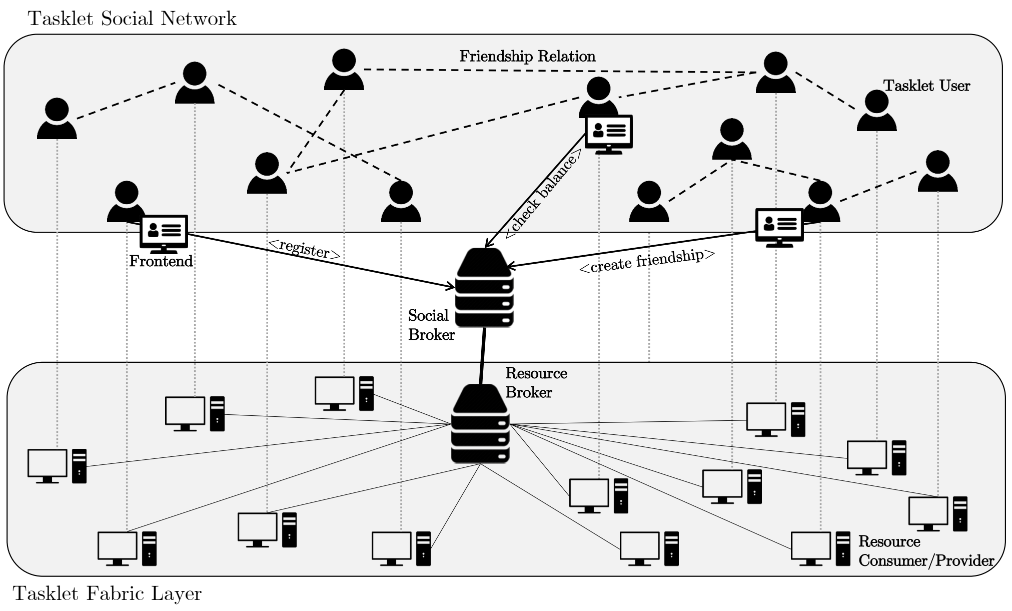
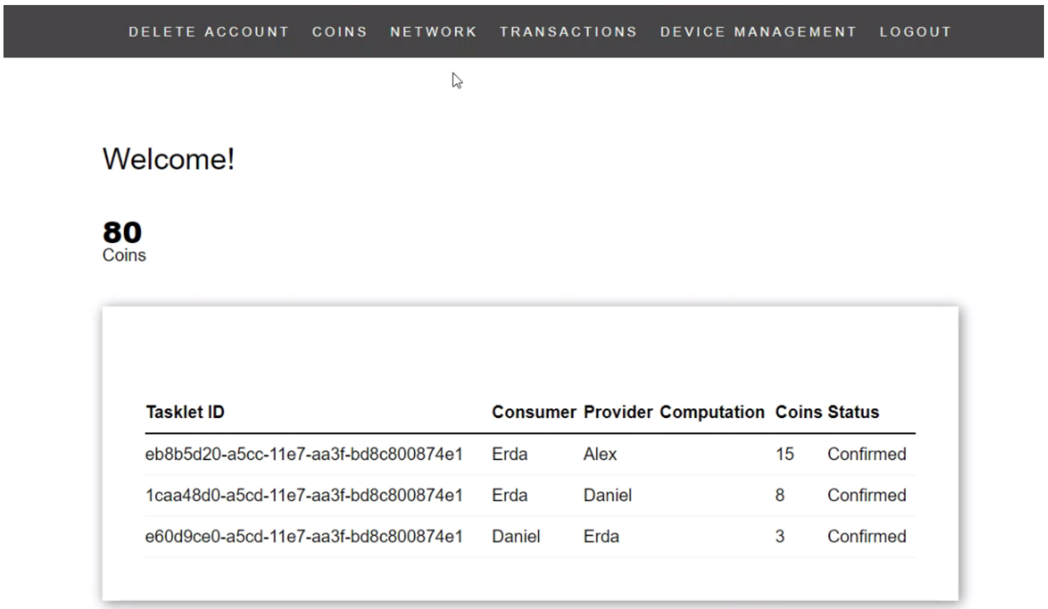


Figure 7.7.: System architecture of the Tasklet social network. The social network is implemented as an additional layer on top of the Tasklet fabric layer which performs the exchange and execution of Tasklets.

with less critical content could be also sent to friends of friends and so on. In combination with the accounting model, friendship relationships can be used to minimize the costs for resource consumers when providers share their resources with friends for free. Transitive friendship relationships might at least result in discounts for Tasklet executions. Thus, both parts of the Tasklet social network interact to implement monetary incentives and meaningful social relationships into the Tasklet system.

One design consideration for the Tasklet social network was to integrate it seamlessly into the existing Tasklet execution system. To avoid confusion, we refer to the Tasklet execution where resource consumers send Tasklets to providers as the *Tasklet fabric layer*. The entities in this fabric layer - consumers, providers, and brokers - should remain independent from the optional social network. Tasklet executions should be possible with or without the social network. Thus, we implemented the Tasklet social network as an overlay on top of the existing Tasklet system as shown in Figure 7.7. The social broker is the central component to store all financial data and social relationships. It keeps track of all user accounts, their balances of the virtual currency, all friendship relationships between users, as well as privacy and pricing settings. In summary, it is the central storage of



Welcome!

**80**
Coins

| Tasklet ID | Consumer | Provider | Computation | Coins | Status |
|---|---|---|---|---|---|
| eb8b5d20-a5cc-11e7-aa3f-bd8c800874e1 | Erda | Alex | | 15 | Confirmed |
| 1caa48d0-a5cd-11e7-aa3f-bd8c800874e1 | Erda | Daniel | | 8 | Confirmed |
| e60d9ce0-a5cd-11e7-aa3f-bd8c800874e1 | Daniel | Erda | | 3 | Confirmed |

Figure 7.8.: Frontend of the Tasklet social network. The current page shows the transaction log which shows all Tasklet executions and the number of coins being transferred.

all user data. The social broker is responsible for any interaction with the users who access the system via the social network frontend which we decribe in the following.

### Frontend

The frontend is implemented as a web application and allows users to manage their accounts and social relationships in an intuitive way. Users can register at the frontend to create an account in the Tasklet social network. In the beginning, their balance is zero and they do not have any friendship connections. A user account consists of four parts. In the *coins* part, users can check their current account balance. They can also request further coins, see all previous coin request, and check the status of their current coin request. Each coin request needs to be approved by a central authority. On approval, the state of the transaction and the balance is updated.

In the *device management* part, users can change the settings of existing devices and add new devices. A user can register any number of devices which all might run Tasklets to increase the balance of the user account. The user can set a different price for each device. After the registration, the user can download the version of the Tasklet execution environment including the Tasklet middleware and the Tasklet virtual machine. In the *transactions* part, users can monitor the current transactions. For each transaction, information about the consumer, the provider, computation cost, and the current status of the transaction are logged. When the Tasklet is executed, the transaction is confirmed and the balances of both users, i.e., the provider and the consumer, get updated. Finally, in the *network* part, the users can browse the Tasklet social network for friends, send friendship request to other users, accept or decline friendship requests, and see the list of current friends. It is also possible to end a friendship. Figure 7.8 shows the frontend on the example of the *transactions* part.

### Tasklet Execution

The Tasklet execution in the fabric layer is independent from the frontend. Tasklets are neither initiated, nor accepted, nor executed in the frontend. Tasklet handling exclusively happens in the Tasklet fabric layer. The broker in the fabric layer might use information from the social broker if available, however, it does not depend on the existence of the social overlay.

Figure 7.9 shows the message of the protocol for a Tasklet execution. Open circles represent messages that have already been implemented in the Tasklet fabric layer. Filled circles show messages that have been implemented for the Tasklet social network. This protocol includes the following steps. ① A consumer sends a resource request to the broker. ❷ The broker forwards the request to the social broker. ❸ The social broker checks the current balance of the consumer and blocks a fixed amount of coins. This amount serves as an insurance for the provider who will get these coins when the consumer does not successfully finish the transaction. The broker further sends updates about friendship relationships to the broker which holds a copy of the friendship database. ④ The broker makes the scheduling decision. Therefore, it takes the QoC information into account. We discuss this scheduling decision in detail below. ⑤ The broker forwards information about the selected provider to the consumer which ⑥ directly sends the Tasklet to the provider. ⑦ The provider executes the Tasklet and ⑧ sends the result to the consumer. ⑨ It also sends an update about the execution to the broker. ❿ The broker informs the social broker about the successful transaction.

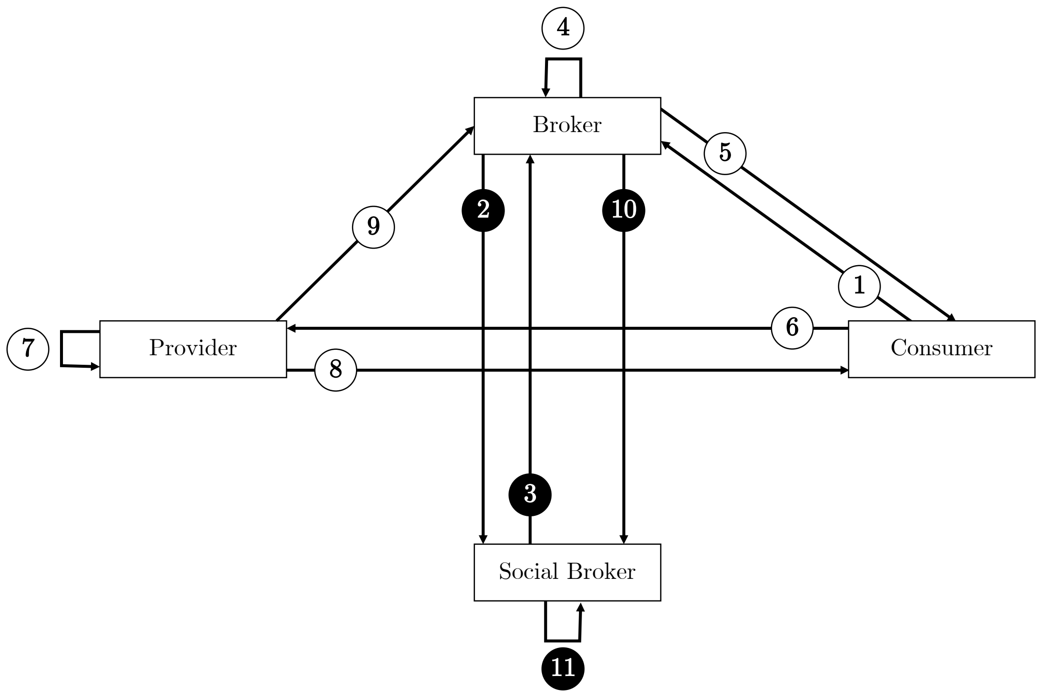


Figure 7.9.: Message flow of a Tasklet execution within the Tasklet social computation sharing system. Open circles represent messages that have already been implemented in the Tasklet fabric layer. Filled circles show messages that have been implemented for the Tasklet social network.

It also notifies the social broker in case an execution was aborted. ⓫ The social broker performs the transaction of coins between the consumer and the provider.

### Data Synchronization

The social broker holds information about all users in the system. This information includes account balances, transaction data, and friendship relationships. In contrast, the broker in the fabric layer holds information about the current status of all providers that are connected to it. Whereas the information in the social network changes less frequently, the live information about the resource providers in the Tasklet broker changes constantly and at a high rate. The scheduling decision is performed on the Tasklet broker in the fabric layer. However, as this decision depends on relationships and prices in the social network, the broker requires information from the social broker when making the scheduling decision. As pricing and friendship information are rather stable, the broker holds a copy of the relevant data to avoid excessive data transfer between the social broker and the Tasklet broker.

There are two possible strategies to synchronize the data between the social broker and the Tasklet broker. First, the brokers could synchronize the social network data whenever the Tasklet broker sends a request to the social broker (❷). In this way, the broker always has the latest information when it makes a scheduling decision. However, the drawback of this approach is that brokers which have not sent a request to the social brokers in a while might need to receive a large amount of data before it can schedule the Tasklet. Second, the brokers could receive updates in given time intervals. While this approach ensures that the updates do not get too large as the updates cannot accumulated over a long time, brokers work based on outdated information between two updates. Thus, we implemented a hybrid approach where the Tasklet broker retrieves an update with every Tasklet request. If there are no Tasklet requests for a given time, the broker receives a periodic update from the social broker. This hybrid approach allows for fast scheduling decisions and preserves data consistency.

### Tasklet Scheduling

The Tasklet fabric layer supports QoC-aware scheduling. Application developers can append these non-functional properties (*QoC goals*) for each Tasklet execution to request a fast, reliable, or confidential execution. The Tasklet social network enhances the scheduling possibilities by introducing friendship relationships. Further, as we introduced accounting into the Tasklet system, in order to minimize the costs for consumers the scheduler also needs to consider the prices that providers charge for the execution of the Tasklet. In the following, we discuss the impact of the additional information from the Tasklet social network on the QoC-aware scheduling.

**Privacy:** The social overlay allows for a finer granularity in the scheduling of confidential Tasklets. Without information about social relationships between resource consumers and providers the only way to enforce privacy is to execute the Tasklet locally on the consumer's device. Direct and transitive friendship connections allow to define multiple levels of privacy ranging from 0 to $n$. A privacy level of 0 means that the Tasklet must not be offloaded but can only be executed locally. Setting the privacy level to 1 allows to execute Tasklets on the devices of direct friends whereas for a privacy level of 2 also friends of friends can be considered as targets. This approach can be extended to an arbitrary level.

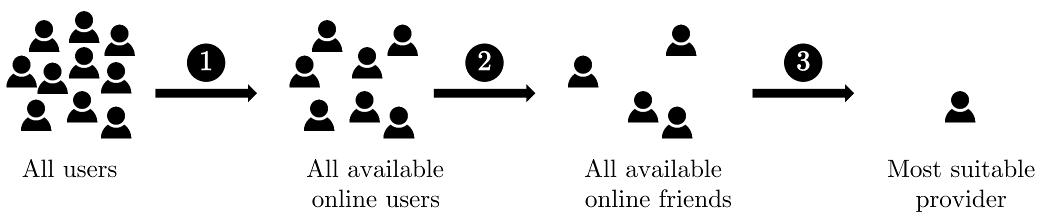


Figure 7.10.: Social scheduling. The scheduler considers only online users with enough idle resources to execute the Tasklet (❶). Among those, only friends are selected in case the Tasklet requires a confidential execution (❷). The scheduler selects the most suitable provider based on a uitlity function (❸).

**Cost:** Resource providers who share their computational power in return for a monetary compensation might not request any payments from their family members or friends. Thus, the Tasklet social network allows to provide discounts to their friends in the network. Further, transitive friendship connections can also receive discounts which might differ from those of direct friends. Similar to the privacy level, this discount policy can be extended to multiple levels.

QoC goals can be conflicting, for example, when the consumer requests a fast but cheap execution. Powerful resources are typically more expensive and cheaper resources, in general, perform worse. In these cases, the scheduler needs to balance the requirements and find the best solution possible. Figure 7.10 shows the different steps of the scheduling decision. ❶ The broker has global knowledge and first filters out all providers that are offline or do not have enough available resources to execute the Tasklet. ❷ Depending on the privacy level, the broker considers only those users that have a direct or transitive friendship relationship with the consumer. ❸ Finally, the most suitable provider is selected. This decision depends on the speed as well as the cost of the provider. To handle this trade-off, consumers can set their priorities which are then used to compute a utility $u$ value for each provider $i$ according to Equation 7.1 where $w_{Cost}$ and $w_{Speed}$ are the weights for costs and speed.

$$u_i = w_{Cost} * \frac{price_i - minPrice}{maxPrice - minPrice} + w_{Speed} * \frac{speed_i - minSpeed}{maxSpeed - minSpeed} \tag{7.1}$$

## Conclusion

The friendship and accounting overlay on top of the Tasklet fabric layer introduces social relationships into the Tasklet system. Instead of only providing the option to share resources with random anonymous users, we give resource owners the control to decide with whom they want to share the computational power of their devices. This should not only increase the willingness to participate in this resource sharing system but also enhances the level of privacy because friendship relations can express a certain level of trust between the users. Further, we are now able to account for Tasklet execution and manage the exchange of computational power and compensation among the users.

# 8. Discussion

In the previous sections, we discussed the Tasklet system, a framework to offload computation to remote devices using Tasklets as a generic abstraction for computation. We discussed the core Tasklet system in Chapter 5 and presented context-aware scheduling approaches for the Tasklet system in Chapter 6. Finally, in Chapter 7, we introduced a social overlay on top of the Tasklet architecture to allow Tasklet accounting and leveraging real-world relationships and incentives to encourage the sharing of computational resources. The design of the Tasklet system, the scheduling strategies, as well as the social overlay has followed the functional and non-functional requirements that we have identified in Chapter 4. Table 8.1 summarizes these requirements. In this chapter, we discuss which requirements are fulfilled by the design and implementation of the Tasklet system and which ones require further attention.

| **Functional Requirements** | **E** | **Nonfunctional Requirements** | **E** |
|---|---|---|---|
| $R_F1$: Task Offloading | ● | $R_{NF}1$: Extensibility | ● |
| $R_F2$: Quality of Service Support | ● | $R_{NF}2$: Scalability | ● |
| $R_F3$: Context-Aware Scheduling | ● | $R_{NF}3$: Performance | ○ |
| $R_F4$: Heterogeneity Support | ● | $R_{NF}4$: Robustness | ● |
| $R_F5$: Accounting | ○ | $R_{NF}5$: Security and Privacy | ○ |
| $R_F6$: Incentives | ● | | |

Table 8.1.: Overview of functional and non-functional requirements. The column E (Evaluation) shows whether each of the requirements is fulfilled.
●: fulfilled, ○: partially fulfilled

## 8.1. Functional Requirements

Here, we discuss which of the functional requirements are fulfilled. These requirements define the core functionalities that the system needs to provide to be usable. It has to be mentioned that even though most functional requirements are fulfilled by the Tasklet system, there is and will always be potential to improve the system and its functionality. The following discussion evaluates to which degree the proof-of-concept Tasklet system implementation in this thesis provides the required functionality.

**$R_F1$ - Task Offloading:** The core functionality of the Tasklet system is to offload computational workload from the local device to remote resources. For this purpose, we have introduced Tasklets, self-contained units of computation that application programmers use to distribute their computational workload. Tasklets contain executable byte code. Resource providers execute this byte code on virtual machines and return computation results to the host application running on the resource consumer. By providing a well-defined programming interface to application programmers which allows them to offload computation and retrieve results in only a few lines of code, task offloading can be integrated into multiple applications independent from the application domain. The offloading process is transparent to both application programmers and application users. Thus, we consider $R_F1$ as completely fulfilled.

**$R_F2$ - Quality of Service Support:** The major motivation for the Tasklet system was to find an abstraction for computation that can be used by as many applications as possible. Since these applications have different requirements in terms of, for example, performance, reliability, and security, this abstraction needs to allow for a flexible handling of computational guarantees that are tailored to the specific needs of an application. In the Tasklet system, we have introduced the concept of Quality of Computation (QoC) that allows application developers to define which guarantees need to be implemented for the application. Therefore, we provide multiple QoC goals such as reliability, speed, or cost which can be set on the level of an individual Tasklet. The Tasklet middleware enforces these goals by means of QoC mechanisms. At runtime, the middleware decides which mechanism or combination of mechanisms is applied. This decision depends on the current context of the computation. This separation between high level goals on the one hand and context-aware enforcement on the other hand makes the quality of service transparent for application developers and users. Thus, $R_F2$ is fulfilled.

**$R_F3$ - Context-Aware Scheduling:** In $R_F1$, we discussed *how* computation can be offloaded to remote devices and introduced the concept of Tasklets. Scheduling is concerned of *where* to offload these Tasklet to and how to find suitable resource providers for execution. This is of particular importance because the Tasklet system can be applied in different computing environments. One environment can either consist of homogeneous, high-performing, reliable cloud instances. Another environment might consist of heterogeneous, low-end, fluctuating edge devices. A third environment can be any mix of both. We have implemented context-aware scheduling in the Tasklet in multiple forms. A centralized Tasklet broker monitors the state of the resource providers and selects only those providers for execution that have idle resources. In Section 6.2, we introduced a fault-avoidant task scheduler that takes the context of the providers, the computation environment, and the computational task into account. In Section 7.3, we introduced a social overlay on top of the core Tasklet system that allows to use social context information to improve privacy and cost efficiency. Even though multiple more context-aware scheduling strategies are possible, we have shown that the integration of these strategies into the Tasklet system is feasible and consider $R_F3$ as fulfilled.

**$R_F4$ - Heterogeneity Support:** Computation offloading systems need to handle multiple forms of heterogeneity such as different programming languages, levels of reliabilities, hardware and software capabilities, and availabilities. The Tasklet system supports multiple programming languages by providing an API on byte array level. Developers can access the Tasklet system by manually translating their offloading requests into the Tasklet format. We provide libraries for Java, C#, and Android applications to automate the marshalling and demarshalling process and make the usage of the Tasklet system as convenient as possible for the user. The Tasklet system handles hardware heterogeneity by providing a lightweight virtual machine for Tasklet execution that runs on a large variety of devices and supports all major platforms. Specialized implementations also make GPUs and embedded devices usable in the Tasklet system. QoC support and context-aware scheduling allow to integrate also unpredictable devices. Thus, the Tasklet system covers multiple forms of heterogeneity as required in $R_F4$.

**$R_F5$ - Accounting:** As Tasklet distribution might not only happen among devices of a single person or company, it is important to track Tasklet execution. This is of particular importance when computation is shared in return of a monetary or non-monetary compensation. The Tasklet system provides a fine-granular tracking of computational resource exchange. The design of the Tasklet virtual machine allows for monitoring the amount of performed computation on the level of a single instruction which can be used as a unit for computation accounting. In Section 7.3, we presented the social computing sharing platform that introduces a social broker which keeps track of all computational exchanges within a Tasklet network. It allows for the mapping of a virtual or real currency to execution cycles and performs the accounting for executed Tasklets between the participants of the Tasklet system. While the basic concept of computation accounting is implemented, there are still open issues to discuss. These open questions include the handling of interrupted Tasklet executions as well as the optimal pricing of resources with respect to their functional and non-functional characteristics such as performance, reliability, and availability. As these questions are part of future work, we consider $R_F5$ as partially fulfilled.

**$R_F6$ - Incentives:** Computational resource sharing systems for end-user devices should provide any form of incentive to participate in resource sharing. As for volunteer computing systems this incentive is of rather intrinsic nature, individuals might not be willing to share their resources with strangers or for-profit companies without any kind of compensation. In Section 7.2, we examined the effect of different types of incentives on the willingness to share. As the findings suggest that this willingness depends on a combination of social relationships and incentive types, we introduced the social overlay on top of the Tasklet system in Section 7.3 that does not only allow computation accounting for $R_F5$ but also gives resource owners the option to request different kinds of compensations for different types of users. Thus, users can still share their resources for free out of altruistic motives but also share for monetary compensation as a result of rather extrinsic motivation. Hence, $R_F6$ is fulfilled.

## 8.2. Nonfunctional Requirements

In contrast to the functional requirements discussed above, the non-functional requirements are not vital for the functioning of a computational resource sharing system. However, they determine whether a system can be successful when applied in real world environments. A system that, for example, is not extensible and cannot grow with changing demands will eventually be replaced by other systems or might not even be applied at all. Thus, non-functional requirements are of no less importance than functional requirements. In the following, we discuss which of these requirements are fulfilled by the Tasklet system.

**$R_{NF}$1 - Extensibility:** Even though we have put a lot of consideration into the design and implementation of the Tasklet system, we are aware that future demands will go beyond the functionality that the system currently provides. Therefore, we have designed the system in an extensible manner which makes it easy to add and replace features later. Currently, developers have to write Tasklet code in our own language C-- which is then executed on our Tasklet virtual machines. However, in principle, Tasklets are agnostic to the kind of code they carry and it would be easy to replace the programming language and execution environment. Further, the Tasklet protocol is versioned and allows to specify a new version at any point in time while still providing backwards compatibility.

Other types of devices, such as wearables, field-programmable gate arrays, or manycore processors can be integrated into the system. Even though the middleware implementation might be device type specific, such as in GPUs, the same protocol can be used. Providing developer support for further programming language integration works by implementing further libraries in addition to the ones already provided. The modular approach of QoC implementation allows for easily specifying further QoC goals which might be enforced by existing or additional QoC mechanisms. The social overlay facilitates incentive engineering and might introduce gamification aspects as well as different kinds of reciprocal sharing economics. Finally, further context-dimensions can be added by replacing the elements of the MAPE cycle introduced in Section 6.2. Due to the modular and open design and implementation of the Tasklet system, we consider $R_{NF}$1 as fulfilled.

**$R_{NF}$2 - Scalability:** From a theoretical perspective, there is no reason why the Tasklet system should not be deployed on every single computation device. However, scalability issues might limit the number of devices and, thus, the number of Tasklets being executed within the system. To increase scalability, the Tasklet system is designed as a hybrid peer-to-peer system where central brokers only facilitate the resource management and resource allocation while the actual exchange of Tasklets and Tasklet results are performed directly between two peers. While this enhances scalability to some extent, the central brokers will - at some point - be overloaded by the number of providers and consumers in the system. Thus, the Tasklet system allows to dynamically spawn new broker instances which will be loosely connected in a peer-to-peer broker overlay. The purpose of the overlay is to manage the size and the composition of the resource pools. To further reduce the load on the brokers, we have developed decentralized scheduling mechanisms (compare Section 6.3) and demand forecasting (compare Section 6.4.3). Both optimizations reduce the number of resource requests from the consumers to the brokers. The Tasklet system is not limited in its geographical size and new brokers can be spawned at any place. The social overlay (compare Section 7.3) is implemented as a web application that dynamically scales with the number of users. Thus, there are no direct limitations to the scalability of the Tasklet system and $R_{NF}2$ is fulfilled.

**$R_{NF}3$ - Performance:** The performance of a distributed computation system can be measured by the decrease of execution times for computationally intensive tasks and the energy that can be saved on the local device. In Section 5.1.6, we introduced multiple optimizations to minimize the execution time for Tasklets. These optimizations focus on the different components of the overall execution time, such as scheduling duration, computation duration, and the time required to return the results. Several of these optimizations are discussed in further detail in Sections 6.1 (QoC), 6.2 (fault-avoidance), 6.3 (decentralized scheduling), 6.4.1 (prioritization), 6.4.2 (resource reservation) and 6.4.3 (demand forecasting).

The Tasklet API itself, that allows developers to parallelize and offload the computationally intensive sections of their applications, leads to large performance gains as shown in the evaluation of the Tasklet prototype in Section 5.3. Besides the execution time, we have also performed optimizations on the communication overhead. Examples are shown in Section 6.2 (fault avoidance) where we have minimized the number of failed attempts and thus the amount of reschedules as well as in Section 6.3 (decentralized scheduling) where we have reduced the communication overhead between consumers and brokers. Regarding the energy consumption, however, we have not yet performed any optimizations. As these optimizations will be implemented in future work, we consider $R_{NF}3$ as partially fulfilled.

**$R_{NF}4$ - Robustness:** A robust system continues working despite the presence of errors within the system [156]. We have discussed fault-tolerance and fault-avoidance in Section 6.2. Robustness in the Tasklet environment means that the overall system continues working when consumers, providers, or brokers fail. In case of consumer failures, we consider the execution of Tasklets as no longer necessary because the application is no longer active and waiting for the results. Provider failures are covered as follows. First of all, it has to be mentioned that the Tasklet system does not necessarily enforce the successful execution of the Tasklets. Instead, the decision whether this guarantee is required is made by the application developers who can set *reliability* as a QoC goal for a Tasklet. When this goal is activated, the Tasklet middleware monitors the execution and re-initiates the Tasklet in case of provider failures or connection losses (compare Section 5.1.5, QoC). More proactive mechanisms to increase the robustness of the system include redundant executions (Section 6.1) and fault-avoidance (Section 6.2).

The most critical failures of components in the Tasklet system are broker failures. In case of a broker failure, resource management and task allocation cannot be performed for the consumers and providers allocated to the faulty broker. Instead, as a recovery mechanism, the peers connect to a different broker within the broker network. Resource requests and Tasklet executions, however, remain valid as the exchange of Tasklets and Tasklet results is performed directly among the peers. To allow for task scheduling even during the short time span between broker failure and reconnection to an alternative broker, we have introduced decentralized scheduling in Section 6.3 that allows consumers to select providers for execution from their locally cached resource lists. As the Tasklet system continues working despite failures in providers and brokers alike, we consider $R_{NF}4$ to be fulfilled.

**$R_{NF}5$ - Security and Privacy:** The distributed nature of computational offloading systems makes these systems vulnerable to multiple types of attacks. Further, confidential data might be revealed intentionally or unintentionally. Our studies have revealed that security and privacy issues are the most relevant obstacles for individuals to participate in resource sharing systems (compare 7.2). To protect resource providers from malicious Tasklet code, the Tasklet virtual machines are sandboxed and do not have any access to the file system. Further, memory access violations are caught by the virtual machines' memory management. Thus, the execution of Tasklets cannot interfere with any other processes on the device.

An additional type of attack that needs to be considered are Byzantine failures where malicious providers send arbitrary results. The QoC goal *majority vote* (compare Section 5.1.5) can be used to execute Tasklets redundantly and perform a majority vote to identify the correct result. The complexity of this mechanism, however, is highly dependent on the type of the results and is beyond the scope of this thesis. Denial of service attacks might be performed on either the broker or the providers. These attacks are not yet handled by the system. Currently, all messages in the system, including Tasklets and Tasklet results, are not encrypted. We have introduced the QoC mechanism *encryption* but the mechanism has not yet been implemented. As to date only a subset of security and privacy concerns are addressed we consider $R_{NF}5$ to be only partially fulfilled.

# 9. Conclusion

In recent years, we could observe two trends that have paved the way for computational resource sharing. First, the number and performance of computational devices have increased. This trend is expected to continue [2, 3]. Second, with the introduction of new types of applications in the field of machine learning, augmented reality, as well as image and video processing, the demand for computation power has increased not only for IT professionals but also for ordinary end users. Computational resource sharing systems allow these applications to offload parts of their work to remote devices. These systems do not only facilitate the mere exchange between tasks and results but support the whole offloading process, including the creation of tasks, the resource management, the allocation of tasks to the remote resources, as well as the result handling. Ideally, developers can use offloading with as minimal effort as possible and the scheduling and distributed execution of tasks is transparent for application developers and users alike.

In the literature, computational resource sharing systems are well covered. There are multiple approaches that support resource sharing for mobile or static devices in grid, cloud, or edge environments. We discussed a selection of 33 approaches in Section 3. Among these systems, only a few made it to maturity and established a regular user base. The successful approaches are volunteer and grid computing systems such as *BOINC* [9] and *HTCondor* [24] that support the execution of scientific projects. Other paradigms such as sharing computing power in a P2P manner among individual device owners have never made it beyond the state of research projects. These observations are in line with the requirements analysis that we performed in Section 4. None of the existing approaches fulfill all requirements for a comprehensive system that would allow to offload task from different types of applications from mobile and static devices to heterogeneous resources in grid, cloud, and edge environments. Instead, the systems focus on single aspects of the problem such as saving energy by code offloading [68, 69], opportunistic resource usage [44, 110], or automated task partitioning [58, 112].

To fill this gap, in this thesis, we proposed our own computational resource sharing system. In Chapter 5, we present the Tasklet system, a best-effort resource sharing middleware that does not only support multiple programming languages but allows for offloading tasks from various types of devices to very different target resources. The Tasklet middleware is a major step towards our vision that computation will become a homogeneous commodity that can be forwarded to all kinds of computational devices regardless of the programming language of the application and system architecture of the host device that created the task.

The Tasklet system provides a best-effort execution layer that allows to execute Tasklets on remote resources but does not provide any execution guarantees. Any further non-functional requirements, such as reliable, fast, or confidential executions, are enforced by our Quality of Computation (QoC) concept that is woven into the Tasklet system. A context-aware scheduler performs the match-making between Tasklets and available resource providers. The scheduler uses the context of the computation environment and the Tasklet itself and selects the most suitable provider for execution. The decoupling of the definition of non-functional requirements for each Tasklet by the application programmer and the enforcement of these guarantees by the Tasklet middleware allows to deploy a wide range of applications in heterogeneous computing environments. We demonstrated multiple context-aware scheduling strategies in Chapter 6.

The Tasklet middleware itself does not consider any social aspects of computational resource sharing. However, as devices are owned by persons who decide how their resources are used, the relationships between resource consumers and providers have to be considered in such a system. Hence, in Chapter 7, we added a social overlay on top of the Tasklet execution middleware that allows users to define with whom they want to share their resources and whether they expect any kind of compensation. The Tasklet scheduler takes these social relationships into account when making scheduling decisions, for example, to keep execution costs low.

The discussion in Chapter 8 shows that the Tasklet system fulfills most of the functional and non-functional requirements. The remaining challenges will be addressed in future work. The Tasklet system has been implemented and tested in real world testbeds with more than 150 heterogeneous physical and virtual devices. This shows that the system is not only valid in theory but can be deployed in practice.

## Outlook

The Tasklet is neither perfect nor completed in its design or implementation. The discussion in Chapter 8 reveals open challenges in the system. First, a comprehensive accounting system is yet to be implemented that handles interrupted executions as well as the pricing for Tasklet executions. Further, the performance optimization, so far, has been focused on the execution speed and reliability. There are further execution parameters that need to be optimized such as energy consumption and costs. Finally, security and privacy concerns have to be addressed, including attacks from malicious users from inside or outside the systems.

Besides the fulfillment of these requirements, we have identified further paths for future research. Currently each Tasklet does not only contain code but also the data that is required for its execution. However, for large datasets this strategy might be suboptimal as it includes a lot of data transfer and might slow down the overall execution. Thus, in the next step, we plan to decouple code and data handling. A first step towards this goal is presented in [323][1].

The Tasklet system already supports multiple types of resources including desktop and laptop computers, mobile devices, and GPUs. In a next step, we plan to include further device types such as microprocessors in embedded devices and FPGAs. This is of particular relevance in the context of the Internet of Things.

Finally, a comprehensive parameter study is required to learn the optimal settings for the Tasklet scheduler in different computing environments. To train the context-aware scheduler, we simulated several scenarios and varied multiple parameters including the composition of the execution environment, the type and number of applications in the system as well as the frequency of Tasklet executions. So far, all optimizations of the system have been implemented and tested independently without paying attention to possible interdependencies among them. Integrating all features directly into the Tasklet middleware would lead to a huge number of scenarios that are infeasible to test in the real world system. However, we plan to perform these parameter studies in a large-scale simulation.

[1] [323] is joint work with M. Breitbach, D. Schäfer, and C. Becker

# Bibliography

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, “SETI@home: an experiment in public-resource computing,” *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[2] D. Evans, “The internet of things: How the next evolution of the internet is changing everything,” *CISCO White Paper*, vol. 1, no. 2011, pp. 1–11, 2011.

[3] M. M. Waldrop, “The chips are down for moore’s law,” *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[4] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, “The case for cyber foraging,” in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. ACM, 2002, pp. 87–92.

[5] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[6] 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, pp. 1–10.

[7] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. Hart, “Enabling multi-tenancy: An industrial experience report,” in *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–8.

[8] 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*, vol. 63, no. 5, pp. 597–610, 2003.

[9] D. P. Anderson, “Boinc: A system for public-resource computing and storage,” in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004, pp. 4–10.

[10] M. van Steen and A. S. Tanenbaum, "A brief introduction to distributed systems," *Computing*, vol. 98, no. 10, pp. 967–1009, 2016.

[11] P. A. Bernstein, "Middleware: a model for distributed system services," *Communications of the ACM*, vol. 39, no. 2, pp. 86–98, 1996.

[12] K. Raymond, "Reference model of open distributed processing (rm-odp): Introduction," in *Open Distributed Processing.* Springer, 1995, pp. 3–14.

[13] N. Sadashiv and S. D. Kumar, "Cluster, grid and cloud computing: A detailed comparison," in *Proceedings of the 6th International Conference on Computer Science & Education (ICCSE).* IEEE, 2011, pp. 477–482.

[14] 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*, vol. 95, 1995, pp. 11–14.

[15] C. Engelmann, H. Ong, and S. L. Scott, "Middleware in modern high performance computing system architectures," in *Proceedings of the International Conference on Computational Science.* Springer, 2007, pp. 784–791.

[16] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 the list," URL: www.top500.de, 2017, Accessed: 2019-01-15.

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

[18] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *The International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[19] M. K. Mishra, Y. S. Patel, M. Ghosh, and G. Mund, "A review and classification of grid computing systems," *International Journal of Computational Intelligence Research*, vol. 13, no. 3, pp. 369–402, 2017.

[20] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien, "Characterizing and evaluating desktop grids: An empirical study," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium.* IEEE, 2004, p. 26.

[21] A. Oram, *Peer-to-Peer: Harnessing the power of disruptive technologies.* O'Reilly Media, Inc., 2001.

[22] S. Choi, H. Kim, E. Byun, M. Baik, S. Kim, C. Park, and C. Hwang, "Characterizing and classifying desktop grid," in *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, 2007, pp. 743–748.

[23] A. A. Chien, S. Marlin, and S. T. Elbert, "Resource management in the entropia system," in *Grid Resource Management.* Springer, 2004, pp. 431–450.

[24] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems.* IEEE, 1988, pp. 104–111.

[25] J. Voas and J. Zhang, "Cloud computing: New wine or just a new bottle?" *IT Professional*, vol. 11, no. 2, pp. 15–17, 2009.

[26] L. Kleinrock, "An internet vision: the invisible global infrastructure," *Ad Hoc Networks*, vol. 1, no. 1, pp. 3–11, 2003.

[27] M. Weiser, "The computer for the 21 st century," *Scientific American*, vol. 265, no. 3, pp. 94–105, 1991.

[28] R. L. Grossman, "The case for cloud computing," *IT Professional*, vol. 11, no. 2, pp. 23–27, 2009.

[29] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[30] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[31] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications.* IEEE, 2008, pp. 825–830.

[32] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *Proceedings of the 5th International Joint Conference on INC, IMS and IDC.* IEEE, 2009, pp. 44–51.

[33] Y. Wang, I.-R. Chen, and D.-C. Wang, "A Survey of Mobile Cloud Computing Applications: Perspectives and Challenges," *Wireless Personal Communications*, vol. 80, no. 4, pp. 1607–1623, 2015.

[34] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, “A survey of mobile cloud computing: architecture, applications, and approaches,” *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[35] M. R. Rahimi, J. Ren, C. H. Liu, A. V. Vasilakos, and N. Venkatasubramanian, “Mobile cloud computing: A survey, state of art and future directions,” *Mobile Networks and Applications*, vol. 19, no. 2, pp. 133–143, 2014.

[36] M. Heck, J. Edinger, D. Schäfer, and C. Becker, “Iot applications in fog and edge computing: Where are we and where are we going?” in *Proceedings of the 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–6.

[37] 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 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE, 2014, pp. 325–329.

[38] S. Sarkar, S. Chatterjee, and S. Misra, “Assessment of the suitability of fog computing in the context of internet of things,” *IEEE Transactions on Cloud Computing*, 2015.

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

[40] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, “Challenges and opportunities in edge computing,” in *IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, 2016, pp. 20–26.

[41] O. Salman, I. Elhajj, A. Kayssi, and A. Chehab, “Edge computing enabling the internet of things,” in *Proceedings of the 2nd World Forum on Internet of Things*. IEEE, 2015, pp. 603–608.

[42] 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*, 2015, pp. 37–42.

[43] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proceedings of the 6th International Conference on Advances in Future Internet.* Citeseer, 2014, pp. 48–55.

[44] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.

[45] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2018.

[46] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[47] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," in *Proceedings of the Global Internet of Things Summit (GIoTS).* IEEE, 2017, pp. 1–6.

[48] A. Ahmed and E. Ahmed, "A Survey on Mobile Edge Computing," in *Proceedings of the 10th IEEE International Conference on Intelligent Systems and Control*, 2016.

[49] 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*, vol. 78, pp. 680–698, 2018.

[50] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing - a key technology towards 5g," *ETSI White Paper*, vol. 11, no. 11, pp. 1–16, 2015.

[51] C. Li, Y. Xue, J. Wang, W. Zhang, and T. Li, "Edge-oriented computing paradigms: A survey on architecture design and system management," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, p. 39, 2018.

[52] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the 1st Edition of the MCC Workshop on Mobile Cloud Computing.* ACM, 2012, pp. 13–16.

[53] I. Stojmenovic, "Fog computing: A cloud to the ground support for smart things and machine-to-machine networks," in *Proceedings of the Telecom-*

*munication Networks and Applications Conference (ATNAC).* IEEE, 2014, pp. 117–122.

[54] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments.* Springer, 2014, pp. 169–186.

[55] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in *Internet of Everything.* Springer, 2018, pp. 103–130.

[56] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[57] J. E. White, "High-level framework for network-based resource sharing," Tech. Rep., 1975.

[58] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th Conference on Computer Systems.* ACM, 2011, pp. 301–314.

[59] 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 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[60] G. Fedak, C. Germain, V. Neri, and F. Cappello, "Xtremweb: A generic global computing system," in *Proceedings of the 1st IEEE/ACM International Symposium onCluster Computing and the Grid.* IEEE, 2001, pp. 582–587.

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

[62] M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," in *Proceedings of the IEEE International Conference on Cloud Engineering.* IEEE, 2014, pp. 57–66.

[63] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 948–960, 1972.

[64] 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*, vol. 28, no. 6, pp. 861–870, 2012.

[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 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2015, pp. 51–60.

[66] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Proceedings of the International Conference on Mobile Computing, Applications, and Services*. Springer, 2010, pp. 59–79.

[67] J. Li, J. Jin, D. Yuan, M. Palaniswami, and K. Moessner, "EHOPES: Data-centered Fog platform for smart living," in *Proceedings of the International Telecommunication Networks and Applications Conference (ITNAC)*, 2015.

[68] 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 8th International Conference on Mobile systems, Applications, and Services*. ACM, 2010, pp. 49–62.

[69] 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 Conference on Computer Communications (INFOCOM)*. IEEE, 2012, pp. 945–953.

[70] H.-Y. Chen, Y.-H. Lin, and C.-M. Cheng, "Coca: Computation offload to clouds using aop," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2012, pp. 466–473.

[71] G. C. Hunt, M. L. Scott *et al.*, "The coign automatic distributed partitioning system," in *Proceedings of the 10th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 99, 1999, pp. 187–200.

[72] G. Chen, B.-T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R. Chandramouli, "Studying energy trade offs in offloading computation/-

compilation in java-enabled mobile devices," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 795–809, 2004.

[73] E. E. Marinelli, "Hyrax: cloud computing on mobile devices using mapreduce," Carnegie-Mellon University Pittsburgh, PA School of Computer Science, Tech. Rep., 2009.

[74] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems.* IEEE, 2002, pp. 217–226.

[75] 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 *et al.*, "Survey on grid resource allocation mechanisms," *Journal of Grid Computing*, vol. 12, no. 2, pp. 399–441, 2014.

[76] J. M. Schopf, "A general architecture for scheduling on the grid," *Special Issue of JPDC on Grid Computing*, vol. 4, 2002.

[77] N. J. Navimipour and F. S. Milani, "A comprehensive study of the resource discovery techniques in peer-to-peer networks," *Peer-to-Peer Networking and Applications*, vol. 8, no. 3, pp. 474–492, 2015.

[78] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[79] L. Ismail, "Dynamic resource allocation mechanisms for grid computing environment," in *Proceedings of the 3rd International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities.* IEEE, 2007, pp. 1–5.

[80] Y. Jiang, "A survey of task allocation and load balancing in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 585–599, 2016.

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

[82] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications," in *Proceedings of the Workshop on Mobile Computing Systems and Applications*. IEEE, 1994, pp. 85–90.

[83] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015.

[84] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[85] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2003, pp. 107–114.

[86] R. Süselbeck, G. Schiele, P. Komarnicki, and C. Becker, "Efficient bandwidth estimation for peer-to-peer systems," in *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*. IEEE, 2011, pp. 10–19.

[87] M. A. Khan, "A survey of computation offloading strategies for performance improvement of applications running on mobile devices," *Journal of Network and Computer Applications*, vol. 56, pp. 28–40, 2015.

[88] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proceedings of the 10th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 12, 2012, pp. 93–106.

[89] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience," *Concurrency Computation Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[90] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[91] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, "Ourgrid: An approach to easily assemble grids with equitable resource sharing," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2003.

[92] C. Vecchiola, R. N. Calheiros, D. Karunamoorthy, and R. Buyya, "Deadline-driven provisioning of resources for scientific applications in hybrid clouds with aneka," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 58–65, 2012.

[93] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[94] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[95] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani, "An open ecosystem for mobile-cloud convergence," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 63–70, 2015.

[96] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, 2015.

[97] S. Abolfazli, Z. Sanaei, M. Shiraz, and A. Gani, "Momcc: market-oriented architecture for mobile cloud computing based on service oriented architecture," in *Proceedings on the 1st IEEE International Conference on Communications in China Workshops (ICCC)*. IEEE, 2012, pp. 8–13.

[98] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Unleashing the power of mobile cloud computing using thinkair," *Computing Research Repository (CoRR)*, vol. abs/1105.3232, 2011.

[99] D. Kovachev, T. Yu, and R. Klamma, "Adaptive computation offloading from mobile devices into the cloud," in *Proceedings of the 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2012, pp. 784–791.

[100] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra, and A. V. Vasilakos, "Mapcloud: mobile applications on an elastic and scalable 2-tier cloud architecture," in *Proceedngs of the 5th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2012, pp. 83–90.

[101] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices,"

in *Proceedings of the 13th ACM International Symposium on Mobile Ad Hoc Networking and Computing.* ACM, 2012, pp. 145–154.

[102] P. Angin and B. K. Bhargava, "An agent-based optimization framework for mobile-cloud computing," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 4, no. 2, pp. 1–17, 2013.

[103] F. Bellifemine, A. Poggi, and G. Rimassa, "Jade: a fipa2000 compliant agent development environment," in *Proceedings of the 5th International Conference on Autonomous Agents.* ACM, 2001, pp. 216–217.

[104] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "Libwater: heterogeneous distributed computing made easy," in *Proceedings of the 27th International ACM Conference on Supercomputing.* ACM, 2013, pp. 161–172.

[105] Y.-W. Kwon and E. Tilevich, "Reducing the energy consumption of mobile applications behind the scenes," in *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2013, pp. 170–179.

[106] H. Qian and D. Andresen, "Jade: An efficient energy-aware computation offloading system with heterogeneous network interface bonding for ad-hoc networked mobile devices," in *Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD).* IEEE, 2014, pp. 1–8.

[107] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy.* IEEE, 2009, pp. 79–93.

[108] Z. Cheng, P. Li, J. Wang, and S. Guo, "Just-in-time code offloading for wearable computing," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 1, pp. 74–83, 2015.

[109] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi, "Kahawai: High-quality mobile gaming using gpu offload," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2015, pp. 121–135.

[110] 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 8th International Conference on Cloud Computing (CLOUD).* IEEE, 2015, pp. 9–16.

[111] D. Huang, L. Yang, and S. Zhang, "Dust: Real-time code offloading system for wearable computing," in *Proceedings of the Global Communications Conference (GLOBECOM).* IEEE, 2015, pp. 1–7.

[112] Y. Li and W. Gao, "Code offload with least context migration in the mobile cloud," in *Proceedings of the Conference on Computer Communications (INFOCOM).* IEEE, 2015, pp. 1876–1884.

[113] G. Orsini, D. Bade, and W. Lamersdorf, "Cloudaware: Towards context-adaptive mobile cloud computing," in *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management.* IEEE, 2015, pp. 1190–1195.

[114] ——, "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, pp. 112–119.

[115] ——, "Cloudaware: A context-adaptive middleware for mobile edge and cloud computing applications," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W).* IEEE, 2016, pp. 216–221.

[116] 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 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering.* IEEE, 2015, pp. 151–156.

[117] R. Friedman and N. Hauser, "Coara: Code offloading on android with aspectj," *Computing Research Repository (CoRR)*, 2016.

[118] P. A. Rego, P. B. Costa, E. F. Coutinho, L. S. Rocha, F. A. Trinta, and J. N. de Souza, "Performing computation offloading on multiple platforms," *Computer Communications*, vol. 105, pp. 1–13, 2017.

[119] A. Banerjee, X. Chen, J. Erman, V. Gopalakrishnan, S. Lee, and J. Van Der Merwe, "Moca: a lightweight mobile cloud offloading architecture," in

*Proceedings of the 8th ACM International Workshop on Mobility in the Evolving Internet Architecture.* ACM, 2013, pp. 11–16.

[120] H. Casanova and J. Dongarra, "Netsovle: A network server for solving computational science problems," in *Proceedings of the ACM/IEEE Conference on Supercomputing.* IEEE, 1996, pp. 40–40.

[121] F. Berman and R. Wolski, "The apples project: A status report," in *Proceedings of the 8th NEC Research Symposium*, vol. 16. Citeseer, 1997.

[122] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama, and R. Buyya, "mcloud: A context-aware offloading framework for heterogeneous mobile cloud," *IEEE Transactions on Services Computing*, vol. 10, no. 5, pp. 797–810, 2017.

[123] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP).* ACM, 2013, pp. 69–84.

[124] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software: Practice and Experience*, vol. 23, no. 12, pp. 1305–1336, 1993.

[125] Amazon.com, Inc., "Amazon EC2 Spot Instances," URL: www.aws.amazon.com/ec2/purchasing-options/spot-instances, Accessed: 2019-01-15.

[126] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid," vol. 1, pp. 283–289, 2000.

[127] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crumme *et al.*, "The grads project: Software support for high-level grid application development," *The International Journal of High Performance Computing Applications*, vol. 15, no. 4, pp. 327–344, 2001.

[128] E. Huedo, I. M. Llorente *et al.*, "The gridway framework for adaptive scheduling and execution on grids," *Scalable Computing: Practice and Experience*, vol. 6, no. 3, 2005.

[129] 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 21st International Symposium on High-Performance Parallel and Distributed Computing.* ACM, 2012, pp. 173–186.

[130] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: computation offloading as a service for mobile devices," in *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing.* ACM, 2014, pp. 287–296.

[131] N. H. Kapadia and J. A. Fortes, "Punch: An architecture for web-enabled wide-area network-computing," *Cluster Computing*, vol. 2, no. 2, pp. 153–164, 1999.

[132] R. L. Henderson, "Job scheduling under the portable batch system," in *Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 1995, pp. 279–294.

[133] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing.* Springer, 2003, pp. 44–60.

[134] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *Proceedings of the 17th international Symposium on High Performance Distributed Computing.* ACM, 2008, pp. 87–96.

[135] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The mars approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, 1989.

[136] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 3590–3605, 2016.

[137] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Transactions on Communications*, vol. 64, no. 10, pp. 4268–4282, 2016.

[138] A. J. Nicholson and B. D. Noble, "Breadcrumbs: forecasting mobile connectivity," in *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking.* ACM, 2008, pp. 46–57.

[139] J. Li, K. Bu, X. Liu, and B. Xiao, "Enda: Embracing network inconsistency for dynamic application offloading in mobile cloud computing," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Mobile Cloud Computing.* ACM, 2013, pp. 39–44.

[140] F. Douglis and J. Ousterhout, "Transparent process migration: Design alternatives and the sprite implementation," *Software: Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.

[141] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, "Javelin: Internet-based parallel computing using java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1139–1160, 1997.

[142] M. O. Neary, A. Phipps, S. Richman, and P. Cappello, "Javelin 2.0: Java-based parallel computing on the internet," in *European Conference on Parallel Processing.* Springer, 2000, pp. 1231–1238.

[143] H. Nakada, M. Sato, and S. Sekiguchi, "Design and implementations of ninf: towards a global computing infrastructure," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 649–658, 1999.

[144] 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, pp. 162–178.

[145] B. N. Chun and D. E. Culler, "Rexec: A decentralized, secure remote execution environment for clusters," in *Proceedings of the International Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing.* Springer, 2000, pp. 1–14.

[146] P. Chandra, Y.-H. Chu, A. Fisher, J. Gao, C. Kosak, T. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Customizable resource management for value-added network services," *IEEE Network*, vol. 15, no. 1, pp. 22–35, 2001.

[147] F. Kon, R. H. Campbell, M. D. Mickunas, K. Nahrstedt, and F. J. Ballesteros, "2k: A distributed operating system for dynamic heterogeneous environments," in *Proceedings of the 9th International Symposium on High-Performance Distributed Computing.* IEEE, 2000, pp. 201–208.

[148] 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 (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 1–10.

[149] ——, "Increasing the efficiency of code offloading through remote-side caching," in *Proceedings of the 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob).* IEEE, 2015, pp. 573–580.

[150] S. Guo, B. Xiao, Y. Yang, and Y. Yang, "Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing," in *Proceedings of the 35th Conference on Computer Communications (INFOCOM).* IEEE, 2016, pp. 1–9.

[151] L. Maciaszek, *Requirements analysis and system design.* Pearson Education, 2007.

[152] P. A. Laplante, *Requirements engineering for software and systems.* Auerbach Publications, 2017.

[153] G. Koelsch, *Requirements writing for system engineering.* Springer, 2016.

[154] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms.* Prentice-Hall, 2007.

[155] B. C. Neuman, "Scale in distributed systems," pp. 463–489, 1994.

[156] J.-C. Fernandez, L. Mounier, and C. Pachon, "A model-based approach for robustness testing," in *Proceedings of the IFIP International Conference on Testing of Communicating Systems.* Springer, 2005, pp. 333–348.

[157] D. Schafer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets:" better than best-effort" computing," in *Proceedings of the 25th International Conference on Computer Communication and Networks (ICCCN).* IEEE, 2016, pp. 1–11.

[158] S. Deering, "Watching the waist of the protocol hourglass," in *Proceedings of the 6th IEEE International Conference on Network Protocols*, 1998.

[159] Google Inc., "Google App Engine: Platform as a Service," URL: www.cloud.google.com/appengine/docs, Accessed: 2019-01-15.

[160] S. Choochotkaew, H. Yamaguchi, T. Higashino, D. Schäfer, J. Edinger, and C. Becker, "Self-adaptive resource allocation for continuous task offloading in pervasive computing," in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops).* IEEE, 2018, pp. 663–668.

[161] 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 27th International Conference on Computer Communication and Networks (ICCCN).* IEEE, 2018, pp. 1–11.

[162] 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 IEEE International Conference on Pervasive Computing and Communications (PerCom).* IEEE, 2017, pp. 79–88.

[163] K. Kumar and Y. H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[164] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *Proceedings of the Conference on Computer Communications (INFOCOM).* IEEE, 2013, pp. 1285–1293.

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

[166] Y. Li and M. Mascagni, "Improving performance via computational replication on a large-scale computational grid." in *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, vol. 3, 2003, p. 442.

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

[168] F. Xhafa and A. Abraham, “Computational models and heuristic methods for grid scheduling problems,” *Future Generation Computer Systems*, vol. 26, no. 4, pp. 608–621, 2010.

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

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

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

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

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

[174] F. Berg, F. Dürr, and K. Rothermel, “Increasing the efficiency and responsiveness of mobile applications with preemptable code offloading,” in *Proceedings of the 2014 IEEE International Conference on Mobile Services*. IEEE, 2014, pp. 76–83.

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

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

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

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

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

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

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

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

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

[184] K. A. Hummel and G. Jelleschitz, "A robust decentralized job scheduling approach for mobile peers in ad-hoc grids," in *Proceedings of the 7th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2007, pp. 461–470.

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

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

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

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

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

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

[191] J. Edinger, D. Schäfer, and C. Becker, "Decentralized scheduling for tasklets," in *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference.* ACM, 2016, pp. 7–8.

[192] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, vol. 10, no. 8, 2010, pp. 89–92.

[193] J.-S. Kim, B. Bhattacharjee, P. J. Keleher, and A. Sussman, "Matching jobs to resources in distributed desktop grid environments," Tech. Rep., 2006.

[194] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proceedings of the 2014 ACM Conference on SIGCOMM.* ACM, 2014, pp. 431–442.

[195] E. Ogston and S. Vassiliadis, "Matchmaking among minimal agents without a facilitator," in *Proceedings of the 5th International Conference on Autonomous Agents.* ACM, 2001, pp. 608–615.

[196] ——, "Local distributed agent matchmaking," in *Proceedings of the International Conference on Cooperative Information Systems.* Springer, 2001, pp. 67–79.

[197] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[198] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[199] J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman, “Resource discovery techniques in distributed desktop grid environments,” in *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*. IEEE Computer Society, 2006, pp. 9–16.

[200] G. Jackson, P. Keleher, and A. Sussman, “Decentralized scheduling and load balancing for parallel programs,” in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2014, pp. 324–333.

[201] J. Lee, P. Keleher, and A. Sussman, “Decentralized dynamic scheduling across heterogeneous multi-core desktop grids,” in *Proceedings of the International Symposium on Parallel & Distributed Processing, Workshops and Phd forum (IPDPSW)*. IEEE, 2010, pp. 1–9.

[202] A. Iamnitchi, I. Foster, and D. Nurmi, “A peer-to-peer approach to resource discovery in grid environments,” in *IEEE High Performance Distributed Computing*, vol. 140, 2002.

[203] The Apache software foundation, “Apache storm,” URL: www.storm.apache.org, Accessed: 2018-04-29.

[204] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, “Distributed qos-aware scheduling in storm,” in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 2015, pp. 344–347.

[205] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-aware operator placement for stream-processing systems,” in *Proceedings of the 22nd International Conference on Data Engineering*. IEEE, 2006, pp. 49–49.

[206] A. Mohaisen, H. Tran, A. Chandra, and Y. Kim, “SocialCloud: Using Social Networks for Building Distributed Computing Services,” *Computing Research Repository (CoRR)*, 2011.

[207] ——, “Trustworthy distributed computing on social networks,” *IEEE Transactions on Services Computing*, vol. 7, no. 3, pp. 333–345, 2014.

[208] J. Edinger, S. VanSyckel, C. Krupitzer, J. M. Paluska, and C. Becker, “Developing a qos-based tasklet trading system,” in *Proceedings of the International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2014, pp. 413–418.

[209] M. Pfannemüller, M. Weckesser, R. Kluge, J. Edinger, M. Luthra, R. Klose, C. Becker, and A. Schürr, “Coalaviz: Supporting traceability of adaptation decisions in pervasive communication systems,” in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2019.

[210] M. Pfannemüller, J. Edinger, M. Weckesser, R. Kluge, M. Luthra, R. Klose, C. Becker, and A. Schürr, “Visualizing adaptation decisions in pervasive communication systems,” in *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Demonstrations (PerCom Demos)*. IEEE, 2019.

[211] S. Choochotkaew, H. Yamaguchi, T. Higashino, M. Shibuya, and T. Hasegawa, “EdgeCEP: Fully-distributed Complex Event Processing on IoT Edges,” in *Proceedings of the IEEE 13th International Conference on Distributed Computing in Sensor Systems*, 2017, pp. 121–129.

[212] H. Li, “Workload characterization, modeling, and prediction in grid computing,” Ph.D. dissertation, Leiden University, 2008.

[213] R. Mayer, B. Koldehofe, and K. Rothermel, “Predictable low-latency event detection with parallel complex event processing,” *IEEE Internet of Things Journal*, vol. 2, no. 4, pp. 274–286, 2015.

[214] O. Nov, D. Anderson, and O. Arazy, “Volunteer computing: a model of the factors determining contribution to community-based scientific research,” in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 741–750.

[215] K. Chard, K. Bubendorfer, S. Caton, and O. F. Rana, "Social cloud computing: A vision for socially motivated resource sharing," *IEEE Transactions on Services Computing*, vol. 5, no. 4, pp. 551–563, 2012.

[216] O. Nov, O. Arazy, and D. P. Anderson, "Technology-Mediated Citizen Science Participation: A Motivational Model," in *Proceedings of the 5th International AAAI Conference on Weblogs and Social Media*, no. July, 2011, pp. 249–256.

[217] ——, "Scientists@Home: What drives the quantity and quality of online citizen science participation?" *PLoS ONE*, vol. 9, no. 4, pp. 1–11, 2014.

[218] D. Ariely, A. Bracha, and S. Meier, "Doing good or doing well? image motivation and monetary incentives in behaving prosocially," *American Economic Review*, vol. 99, no. 1, pp. 544–55, 2009.

[219] M. Bashir, B. Strickland, and J. Bohr, "What motivates people to use Bitcoin?" in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10047 LNCS. Springer, Cham, 2016, pp. 347–367.

[220] J. Carpenter and C. K. Myers, "Why volunteer? Evidence on the role of altruism, image, and incentives," *Journal of Public Economics*, vol. 94, no. 11-12, pp. 911–920, 2010.

[221] S. Dellavigna, J. A. List, and U. Malmendier, "Testing for altruism and social pressure in charitable giving," *Quarterly Journal of Economics*, vol. 127, no. 1, pp. 1–56, 2012.

[222] P. Golle, K. Leyton-Brown, I. Mironov, and M. Lillibridge, "Incentives for sharing in peer-to-peer networks," in *Proceedings of the International Workshop on Electronic Commerce.* Springer, 2001, pp. 75–87.

[223] C. Haas, S. Caton, K. Chard, and C. Weinhardt, "Co-operative infrastructures: An economic model for providing infrastructures for social Cloud Computing," in *Proceedings of the 46th Annual Hawaii International Conference on System Sciences*, 2013, pp. 729–738.

[224] C. Haas, *Incentives and two-sided matching: Engineering coordination mechanisms for social clouds*, 2014.

[225] F. Hawlitschek, T. Teubner, and H. Gimpel, "Understanding the sharing economy - Drivers and impediments for participation in peer-to-peer rental," in *Proceedings of the 49th Annual Hawaii International Conference on System Sciences*, vol. 2016-March. IEEE, 2016, pp. 4782–4791.

[226] A. Holohan and A. Garg, "Collaboration Online: The Example of Distributed Computing," *Journal of Computer-Mediated Communication*, vol. 10, no. 4, 2005.

[227] G. Iosifidis, L. Gao, J. Huang, and L. Tassiulas, "Incentive mechanisms for user-provided networks," *IEEE Communications Magazine*, vol. 52, no. 9, pp. 20–27, 2014.

[228] K. John, K. Bubendorfer, and K. Chard, "A social cloud for public eResearch," in *Proceedings of the 7th IEEE International Conference on eScience*, 2011, pp. 363–370.

[229] J. Hamari and J. Koivisto, "Why do people use gamification services?" *International Journal of Information Management*, vol. 35, no. 4, pp. 419–431, 2015.

[230] M. Punceva, I. Rodero, M. Parashar, O. F. Rana, and I. Petri, "Incentivising resource sharing in social clouds," *Concurrency Computation*, vol. 27, no. 6, pp. 1483–1497, 2015.

[231] D. Toth, R. Mayer, and W. Nichols, "Increasing participation in volunteer computing," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1878–1882.

[232] B. Butler, "Membership size, communication activity and sustainability: The internal dynamics of networked social structures," *Information Systems Research*, vol. 12, no. July 2015, pp. 346–362, 2001.

[233] S. Caton, C. Dukat, T. Grenz, C. Haas, M. Pfadenhauer, and C. Weinhardt, "Foundations of trust: Contextualising trust in social clouds," in *Proceedings of the 2nd International Conference on Cloud and Green Computing and 2nd International Conference on Social Computing and its Applications*, 2012, pp. 424–429.

[234] S. Caton, C. Haas, K. Chard, K. Bubendorfer, and O. F. Rana, "A social compute cloud: Allocating and sharing infrastructure resources via social

networks," *IEEE Transactions on Services Computing*, vol. 7, no. 3, pp. 359–372, 2014.

[235] R. Gracia-Tinedo, M. Sánchez-Artigas, A. Ramírez, A. Moreno-Martínez, X. León, and P. García-López, "Giving form to social cloud storage through experimentation: Issues and insights," *Future Generation Computer Systems*, vol. 40, pp. 1–16, 2014.

[236] A. Ardichvili, V. Page, and T. Wentling, "Motivation and Barriers to Participation in Virtual Knowledge Sharing Communities of Practice," *Journal of Knowledge Management*, vol. 7, no. 1, pp. 64–77, 2003.

[237] A. Eveleigh, C. Jennett, A. Blandford, P. Brohan, and A. L. Cox, "Designing for dabblers and deterring drop-outs in citizen science," in *Proceedings of the 32nd Annual ACM Conference on Human factors in Computing Systems (CHI)*. ACM Press, 2014, pp. 2985–2994.

[238] R. M. Ryan and E. L. Deci, "Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions," *Contemporary Educational Psychology*, vol. 25, pp. 54–67, 2000.

[239] N. Andrade, F. Brasileiro, W. Cirne, and M. Mowbray, "Discouraging free riding in a peer-to-peer CPU-sharing grid," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*. IEEE, 2004, pp. 129–137.

[240] J. Andreoni, "Impure altruism and donations to public goods: A theory of warm-glow giving," *The Economic Journal*, vol. 100, no. 401, pp. 464–477, 1990.

[241] "BOINCstats," URL: www.boincstats.com, Accessed: 2019-03-16.

[242] A. M. Rashid, K. Ling, R. D. Tassone, P. Resnick, R. Kraut, and J. Riedl, "Motivating participation by displaying the value of contribution," *ACM Conference on Human Factors in Computing Systems*, pp. 955 – 958, 2006.

[243] C. D. Batson, *The altruism question: Toward a social-psychological answer.* Psychology Press, 2014.

[244] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: defining gamification," in *Proceedings of the 15th*

*International Academic MindTrek Conference: Envisioning Future Media Environments.* ACM, 2011, pp. 9–15.

[245] A. Bowser, D. Hansen, Y. He, C. Boston, M. Reid, L. Gunnell, and J. Preece, "Using gamification to inspire new citizen science volunteers," in *Proceedings of the 1st International Conference on Gameful Design, Research, and Applications.* ACM Press, 2013, pp. 18–25.

[246] P. Clary, E.; Snyder, M.; Ridge, R.; Copeland, J.; Stukas, A.; Haugen, J.; Miene, "Personality Processes and individual differences. Understanding and assessing the motivations of volunteers: a functional approach." *Journal of Personality and Social Psychology*, vol. 74, no. 6, pp. 1516–1530, 1998.

[247] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness," in *Proceedings of the 15th International Academic MindTrek Conference on Envisioning Future Media Environments (MindTrek).* ACM Press, 2011, p. 9.

[248] J. Hamari and J. Koivisto, "Social motivations to use gamification: an empirical study of gamifying exercise," *Proceedings of the 21st European Conference on Information Systems*, no. JUNE, pp. 1–12, 2013.

[249] J. Hamari, J. Koivisto, and H. Sarsa, "Does gamification work? - A literature review of empirical studies on gamification," *Proceedings of the 47th Annual Hawaii International Conference on System Sciences*, 2014.

[250] N. Prestopnik and K. Crowston, "Purposeful Gaming & Socio-Computational Systems : A Citizen Science Design Case," *Proceedings of the 17th ACM International Conference on Supporting Group Work*, pp. 75–84, 2012.

[251] L. Festinger, "A theory of social comparison processes," *Human Relations*, vol. 7, no. 2, pp. 117–140, 1954.

[252] A. Bogliolo, P. Polidori, A. Aldini, W. Moreira, P. Mendes, M. Yildiz, C. Ballester, and J. M. Seigneur, "Virtual currency and reputation-based cooperation incentives in user-centric networks," in *Proceedings of the 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2012, pp. 895–900.

[253] H. Zhao, X. Yang, and X. Li, “An incentive mechanism to reinforce truthful reports in reputation systems,” *Journal of Network and Computer Applications*, vol. 35, no. 3, pp. 951–961, 2012.

[254] J. A. Roberts, I.-H. Hann, and S. A. Slaughter, “Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects,” *Management Science*, vol. 52, no. 7, pp. 984–999, 2006.

[255] C. H. Declerck, C. Boone, and G. Emonds, “When do people cooperate? The neuroeconomics of prosocial decision making,” *Brain and Cognition*, vol. 81, no. 1, pp. 95–117, 2013.

[256] R. Belk, “Sharing,” *Journal of Consumer Research*, vol. 36, no. 5, pp. 715–734, 2010.

[257] P. Kollock, “The economies ol online cooperation,” *Communities in Cyberspace*, vol. 220, 1999.

[258] A. M. Khan, Ü. C. Büyükşahin, and F. Freitag, “Incentive-based resource assignment and regulation for collaborative cloud services in community networks,” in *Journal of Computer and System Sciences*, vol. 81, no. 8, 2015, pp. 1479–1495.

[259] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat, “Resource allocation in federated distributed computing infrastructures,” in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure*, vol. 9, 2004.

[260] L. Buttyan and J.-P. Hubaux, “Nuglets: a virtual currency to stimulate cooperation in self-organized mobile ad hoc networks,” Tech. Rep., 2001.

[261] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta, “Spawn: A Distributed Computational Economy,” *IEEE Transactions on Software Engineering*, vol. 18, no. 2, pp. 103–117, 1992.

[262] J. M. Seigneur, J. Abendroth, and C. D. Jensen, “Bank Accounting and Ubiquitous Brokering of Trustos,” in *Proceedings of the 7th Cabernet Radicals Workshop*, 2002, pp. 12–17.

[263] K. Chard, S. Caton, O. Rana, and K. Bubendorfer, "Social cloud: Cloud computing in social networks," in *Proceedings of the 3rd International Conference on Cloud Computing (CLOUD).* IEEE, 2010, pp. 99–106.

[264] 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*, vol. 14, no. 13-15, pp. 1507–1542, 2002.

[265] M. Feldman and J. Chuang, "Overcoming free-riding behavior in peer-to-peer systems," *ACM SIGecom Exchanges*, vol. 5, no. 4, pp. 41–50, 2005.

[266] M. Conti and M. Kumar, "Opportunities in opportunistic computing," *Computer*, vol. 43, no. 1, 2010.

[267] J. Topolsky, "Folding home recognized by guinness world records," URL: www.guinnessworldrecords.com/world-records/most-powerful-distributed-computing-network, 2007, Accessed: 2019-03-16.

[268] 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 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE, 2009, pp. 1–8.

[269] Facebook, "Progress thru processors," 2011.

[270] G. Woltman and S. Kurowski, "The great internet mersenne prime search," URL: www.mersenne.org, Accessed: 2019-03-16.

[271] T. MacWilliam and C. Cecka, "Crowdcl: Web-based volunteer computing with webcl," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC).* IEEE, 2013, pp. 1–6.

[272] B. Cohen, "Incentives Build Robustness in BitTorrent," *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, 2003.

[273] C. Teixeira, R. Azevedo, J. S. Pinto, and T. Batista, "User provided cloud computing," *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pp. 727–732, 2010.

[274] L. Buttyan and J.-P. Hubaux, "Nuglets: a virtual currency to stimulate cooperation in self-organized mobile ad hoc networks," 2001.

[275] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[276] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, 2014.

[277] N. Nisan, S. London, O. Regev, and N. Camiel, “Globally distributed computation over the internet-the popcorn project,” in *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1998, pp. 592–601.

[278] O. Regev and N. Nisan, “The popcorn market. online markets for computational resources,” *Decision Support Systems*, vol. 28, no. 1-2, pp. 177–189, 2000.

[279] A. M. Khan, Ü. C. Büyükşahin, and F. Freitag, “Towards incentive-based resource assignment and regulation in clouds for community networks,” in *Proceedings of the International Conference on Grid Economics and Business Models*. Springer, 2013, pp. 197–211.

[280] Z. Ali, R. U. Rasool, and P. Bloodsworth, “Social networking for sharing cloud resources,” in *Proceedings of the 2nd International Conference on Cloud and Green Computing and 2nd International Conference on Social Computing and its Applications*, 2012, pp. 160–166.

[281] M. Pitkänen, T. Kärkkäinen, J. Ott, M. Conti, A. Passarella, S. Giordano, D. Puccinelli, F. Legendre, S. Trifunovic, K. Hummel *et al.*, “Scampi: service platform for social aware mobile and pervasive computing,” in *Proceedings of the 1st Edition of the MCC Workshop on Mobile Cloud Computing*. ACM, 2012, pp. 7–12.

[282] A. McMahon and V. Milenkovic, “Social Volunteer Computing,” *Journal on Systemics, Cybernetics and Informatics*, vol. 9, no. 4, 2011.

[283] C. Haas, S. Caton, and C. Weinhardt, “Engineering incentives in social clouds,” in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011, pp. 572–575.

[284] A. P. Fiske, “The four elementary forms of sociality: framework for a unified theory of social relations.” *Psychological Review*, vol. 99, no. 4, p. 689, 1992.

[285] J. Heyman and D. Ariely, “Effort for payment: A tale of two markets,” *Psychological Science*, vol. 15, no. 11, pp. 787–793, 2004.

[286] B. S. Frey, "How intrinsic motivation is crowded out and in," *Rationality and Society*, vol. 6, no. 3, pp. 334–352, 1994.

[287] B. S. Frey and R. Jegen, "Motivation crowding theory," *Journal of Economic Surveys*, vol. 15, no. 5, pp. 589–611, 2001.

[288] K. Aquino, I. Reed *et al.*, "The self-importance of moral identity." *Journal of personality and social psychology*, vol. 83, no. 6, p. 1423, 2002.

[289] J. Edinger, L. M. Edinger-Schons, D. Schäfer, A. Stelmaszczyk, and C. Becker, "Of money and morals-the contingent effect of monetary incentives in peer-to-peer volunteer computing," in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[290] M. Buhrmester, T. Kwang, and S. D. Gosling, "Amazon's mechanical turk: A new source of inexpensive, yet high-quality, data?" *Perspectives on Psychological Science*, vol. 6, no. 1, pp. 3–5, 2011.

[291] R. Bénabou and J. Tirole, "Incentives and prosocial behavior," *American Economic Review*, vol. 96, no. 5, pp. 1652–1678, 2006.

[292] B. S. Frey and F. Oberholzer-Gee, "The cost of price incentives: An empirical analysis of motivation crowding-out," *The American Economic Review*, vol. 87, no. 4, pp. 746–755, 1997.

[293] K. Aquino, D. Freeman, A. Reed II, V. K. Lim, and W. Felps, "Testing a social-cognitive model of moral behavior: the interactive influence of situations and moral identity centrality," *Journal of Personality and Social Psychology*, vol. 97, no. 1, p. 123, 2009.

[294] K. Aquino, A. Reed, M. M. Stewart, and D. L. Shapiro, "Self-regulatory identity theory and reactions toward fairness enhancing organizational policies," *What Motivates Fairness in Organizations*, pp. 129–148, 2005.

[295] D. K. Lapsley and D. Narvaez, "Moral development, self and identity: Essays in honor of augusto blasi," 2004.

[296] A. Blasi, "Moral functioning: Moral understanding and personality," *Moral Development, Self, and Identity*, pp. 335–347, 2004.

[297] S. A. Haslam, *Psychology in organizations.* Sage, 2004.

[298] L. M. Edinger-Schons, J. Sipilä, S. Sen, G. Mende, and J. Wieseke, “Are two reasons better than one? the role of appeal type in consumer responses to sustainable products,” *Journal of Consumer Psychology*, 2018.

[299] L. Ang, C. Dubelaar, and B.-C. Lee, “To trust or not to trust? a model of internet trust from the customer’s point of view,” in *Proceedings of the 14th Bled Electronic Commerce Conference*, 2001, pp. 40–52.

[300] Y. E. Riyanto and Y. X. Jonathan, “Directed trust and trustworthiness in a social network: An experimental investigation,” *Journal of Economic Behavior & Organization*, vol. 151, pp. 234–253, 2018.

[301] R. Xiang, J. Neville, and M. Rogati, “Modeling relationship strength in online social networks,” in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 981–990.

[302] M. McPherson, L. Smith-Lovin, and J. M. Cook, “Birds of a feather: Homophily in social networks,” *Annual Review of Sociology*, vol. 27, no. 1, pp. 415–444, 2001.

[303] C.-J. Lin, Y.-T. Chang, S.-C. Tsai, and C.-F. Chou, “Distributed social-based overlay adaptation for unstructured p2p networks,” in *Proceedings of the IEEE Global Internet Symposium*. IEEE, 2007, pp. 1–6.

[304] H. Louch, “Personal network integration: transitivity and homophily in strong-tie relations,” *Social Networks*, vol. 22, no. 1, pp. 45–64, 2000.

[305] E. Gilbert and K. Karahalios, “Predicting tie strength with social media,” in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 2009, pp. 211–220.

[306] V. Podobnik, D. Striga, A. Jandras, and I. Lovrek, “How to calculate trust between social network users?” in *Proceedings of the 20th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2012, pp. 1–6.

[307] X. Li, Q. Yang, X. Lin, S. Wu, and M. Wittie, “Itrust: interpersonal trust measurements from social interactions,” *IEEE Network*, vol. 30, no. 4, pp. 54–58, 2016.

[308] T. A. Snijders, “Statistical models for social networks,” *Annual Review of Sociology*, vol. 37, pp. 131–153, 2011.

[309] A. Rapoport, "Spread of information through a population with socio-structural bias: I. assumption of transitivity," *The Bulletin of Mathematical Biophysics*, vol. 15, no. 4, pp. 523–533, 1953.

[310] F. J. Flynn, R. E. Reagans, and L. Guillory, "Do you two know each other? transitivity, homophily, and the need for (network) closure." *Journal of Personality and Social Psychology*, vol. 99, no. 5, p. 855, 2010.

[311] E. Adar and B. A. Huberman, "Free riding on gnutella," *First Monday*, vol. 5, no. 10, 2000.

[312] D. Hughes, G. Coulson, and J. Walkerdine, "Free riding on gnutella revisited: the bell tolls?" *IEEE Distributed Systems Online*, vol. 6, no. 6, 2005.

[313] P. Ernstberger, "Linden dollar and virtual monetary policy," 2009.

[314] PayPal, "Paypal merchant fees," URL: www.paypal.com/us/webapps/mpp/merchant-fees, Accessed: 2019-03-16.

[315] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer, "Karma: A secure economic framework for peer-to-peer resource sharing," in *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, vol. 35, no. 6, 2003.

[316] D. Hausheer, N. C. Liebau, A. Mauthe, R. Steinmetz, and B. Stiller, "Token-based accounting and distributed pricing to introduce market mechanisms in a peer-to-peer file sharing scenario," in *Proceedings of the 3rd International Conference on Peer-to-Peer Computing.* IEEE, 2003, pp. 200–201.

[317] C. Ondrejka, "Collapsing geography (second life, innovation, and the future of national power)," *Innovations: Technology, Governance, Globalization*, vol. 2, no. 3, pp. 27–54, 2007.

[318] W. Vickrey, "Counterspeculation, auctions, and competitive sealed tenders," *The Journal of Finance*, vol. 16, no. 1, pp. 8–37, 1961.

[319] R. Wolski, J. Brevik, J. S. Plank, and T. Bryan, "Grid resource allocation and control using computational economies," *Grid Computing: Making the Global Infrastructure a Reality*, vol. 772, 2003.

[320] M. Caramia and S. Giordani, "Resource allocation in grid computing: an economic model," *WSEAS Transactions on Computer Research*, vol. 3, no. 1, pp. 19–27, 2008.

[321] K. Wei, A. J. Smith, Y.-F. Chen, and B. Vo, "Whopay: A scalable and anonymous payment system for peer-to-peer environments," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. IEEE, 2006, pp. 13–13.

[322] A. Barmouta and R. Buyya, "Gridbank: A grid accounting services architecture (gasa) for distributed systems sharing and integration," in *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, 2003.

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

[324] E. F. Brickell, C. D. Hall, J. F. Cihula, and R. Uhlig, "Method of improving computer security through sandboxing," 2011, US Patent 7,908,653.

[325] J. A. Gliem and R. R. Gliem, "Calculating, Interpreting, and Reporting Cronbach's Alpha Reliability Coefficient for Likert-Type Scales," in *Midwest Research-to-Practice Conference in Adult, Continuing, and Community Education*, no. 1992, 2003, pp. 82–88.

[326] V. Venkatesh, "Determinants of Perceived Ease of Use: Integrating Control, Intrinsic Motivation, and Emotion into the Technology Acceptance Model," *Information Systems Research*, vol. 11, no. 4, pp. 342–365, 2000.

[327] P. Pavlou A., H. Liang, and Y. Xue, "Understanding and Mitigating Uncertainty in Online Exchange Relationships: A Principal- Agent Perspective," *MIS Quarterly*, vol. 31, no. 1, pp. 105–136, 2007.

# Appendix

## A. Questionnaire

*Dear participants, Thank you very much for taking part in this scientific survey. By participating in our survey you substantially contribute to our research. Of course all your given information will be treated completely anonymously and will not be shared with third parties. Answering the questionnaire will take approximately 15 minutes. There are no right or wrong answers, we appreciate your personal opinion and realistic behavior. We greatly appreciate your participation and your support! Thank you very much! Best regards, Janick Edinger and Laura Marie Edinger-Schons*

*Sharing of computational resources means that people allow others to use the computational power of their own devices such as personal computers or notebooks. For example, imagine you want to render an image or calculate a route to a destination on your smartphone. A special software can manage to offload the task from one device (such as your smartphone) to another device where the computational task is executed. Afterwards, the result is sent back to your smartphone. This can be helpful when your smartphone has less computational power than the task requires or to save your smartphone's battery.*

*Do you own a smartphone?*

- **Yes**
- **No**

*How intensively do you engage in the following activities on your smartphone?*

- **Email**
- **Social media**
- **Watching videos**

- **Reading news**
- **Download applications**
- **Online shopping**
- **Video calling**
- **Streaming music**
- **Online payment**
- **Navigation**
- **Online gaming**
- **Taking pictures**
- **Photo/Video editing**

*Have you ever felt that an app would be faster if your phone had more computational resources?*

- **Never (1)**
- **Very often (7)**

*Do you think that the limited computational resources on your smartphone pose a problem for you?*

- **Not a all (1)**
- **Totally (7)**

*Have you ever felt that the battery of your smartphone drained faster because of a computational intensive app (e.g., navigation, image editing, speech recognition)?*

- **Never (1)**
- **Very often (7)**

*Sharing of computational resources can work in both directions. That means that you can use resources from others and also provide your own resources (e.g., from your personal computer or your laptop) to others. Some systems already exist through which you can share your resources with either private persons or scientific projects (such as medical research that requires large amounts of computational resources). Have you been aware of the possibility to share your own resources with others?*

- **Yes**
- **No**

*Have you ever shared your computational resources with others?*

- **Yes, namely in...**
- **No**

*In how far do you agree with the following statements? (I do not agree at all/ I fully agree)*

- **My experiences with these systems were positive.**
- **Would you be willing to share your computational resources with others?**
- **I would share my resources with others.**

*Now, we would like to ask you to read the following scenario and try to put yourself in the described situation. Group 1: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with friends. The sharing of resources is based on a system of monetary compensation, i.e., you receive money for the computational resources which you share with the other users in the system and you pay for the resources that you use. The software package is further designed as a game in which you compete with other users for the highest score. You can increase your score by providing resources to others. On a leaderboard you can see your performance compared to those of other users. Each day the best-performing contributor receives a symbolic award and earns extra points for the leaderboard.*

*Group 2: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with friends. The sharing of resources is based on a system of monetary compensation, i.e., you receive money for the computational resources which you share with the other users in the system and you pay for the resources that you use. The software package is further designed to transparently track your provision of resources. On a statistics page you can continuously track the amount of resources that you provided to other users and the amount of resources of others that you have used.*

*Group 3: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with friends. The sharing* *of resources is based on a system of monetary compensation, i.e., you receive money for the computational resources which you share with the other users in the system and you pay for the resources that you use.*

*Group 4: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with friends. The sharing of resources is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free. The software package is further designed as a game in which you compete with other users for the highest score. You can increase your score by providing resources to others. On a leaderboard you can see your performance compared to those of other users. Each day the best-performing contributor receives a symbolic award and earns extra points for the leaderboard.*

*Group 5: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with friends. The sharing of resources is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free. The software package is further designed to transparently track your provision of resources. On a statistics page you can continuously track the amount of resources that you provided to other users and the amount of resources of others that you have used.*

*Group 6: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with friends. The sharing of resources is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free.*

*Group 7: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with anonymous users. The sharing of resources is based on a system of monetary compensation, i.e., you receive money for the computational resources which you share with the other users in the system and you pay for the resources that you use. The software package is further designed as a game in which you compete with other users for the highest score. You can increase your score by providing resources to others. On a leaderboard you can see your performance compared to those of other users.*

*Each day the best-performing contributor receives a symbolic award and earns extra points for the leaderboard.*

*Group 8: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with anonymous users. The sharing of resources is based on a system of monetary compensation, i.e., you receive money for the computational resources which you share with the other users in the system and you pay for the resources that you use. The software package is further designed to transparently track your provision of resources. On a statistics page you can continuously track the amount of resources that you provided to other users and the amount of resources of others that you have used.*

*Group 9: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with anonymous users. The sharing of resources is based on a system of monetary compensation, i.e., you receive money for the computational resources which you share with the other users in the system and you pay for the resources that you use.*

*Group 10: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with anonymous users. The sharing of resources is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free. The software package is further designed as a game in which you compete with other users for the highest score. You can increase your score by providing resources to others. On a leaderboard you can see your performance compared to those of other users. Each day the best-performing contributor receives a symbolic award and earns extra points for the leaderboard.*

*Group 11: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with anonymous users. The sharing of resources is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free. The software package is further designed to transparently track your provision of resources. On a statistics page you can continuously track the amount of resources that you provided to other users and the amount of resources of others that you have used.*

*Group 12: Imagine a system that allows you to safely share computational resources. Via this software you can share computational resources with anonymous users. The sharing of resources is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and may use their resources for free.*

*How likely would you be to use such a system?*

- **Not a all likely (1)**
- **Very likely (7)**

*In how far do you agree with the following statements? (I do not agree at all/ I fully agree)*

- **In the software package, I could share computational resources with various friends.**
- **In the software package, I could share computational resources with various anonymous users.**
- **The sharing of resources in the software package is based on a system of monetary compensation, i.e., you receive money for the computational resources you share with the other users in the system and you pay for the resources that you use.**
- **The sharing of resources in the software package is based on a system of reciprocity, i.e., you provide your computational resources to the other users in the system and you can use their resources for free.**
- **The software package is designed as a game in which you compete with other users for the highest score. You can increase your score by providing resources to others. On a leaderboard you can see your performance compared to those of other users. Each day the best-performing contributor receives a symbolic award and earns extra points for the leaderboard.**
- **The software package is designed to transparently track your provision of resources. On a statistics page you can continuously track the amount of resources that you provided to other users and the amount of resources of others that you have used.**

*Which of the following reasons could hinder you from sharing your resources?*

- **I would be worried about security issues.**
- **I would be worried about data privacy.**
- **I was not aware of the possibility to participate.**
- **I do not see any benefits for me.**
- **I do not want my device to slow down.**
- **I think installing such a system is too much effort.**
- **Other:**

*Imagine a system that allows you to safely share your computational resources with others. How likely would you be to share your resources with the following users?*

- **Friends and family**
- **Anonymous private users**
- **Scientific projects**
- **Non-profit organizations**
- **For-profit organizations**

*Now, imagine such a system where device owners can be compensated for sharing their resources. They can be compensated in multiple ways: They could share their resources for free, which means that they do not receive a compensation. They could receive a compensation to cover their energy costs. They could receive a compensation to make a profit. Which compensation would you demand from the following users for sharing your resources?*

*Users:*

- **Friends and family**
- **Anonymous private users**
- **Scientific projects**
- **Non-profit organizations**
- **For-profit organizations**

*Compensation:*

- **For free**
- **Cover costs**
- **Be profitable**
- **I would not share my resources**

*Would you be interested in a system where you can share your resources in a tit-for-tat manner that means when you allow others to use your resources you can access their resources in return?*

- **Not interested at all (1)**
- **Very interested (7)**

In general, what would be your motivation to participate in such a system?

- **To help others.**
- **To use resources more efficiently.**
- **To contribute to scientific projects.**
- **To get access to more resources myself.**
- **To earn money.**
- **Others:**

*Currently, the field of virtual currencies (e.g., Bitcoins) became very popular. Virtual currencies have to be created ("mined") by solving complex computational tasks. This typically requires large computational resources. Would you engage in mining virtual currencies (such as mining Bitcoins) to earn money?*

- **Definitely not (1)**
- **Definitely yes (7)**

In how far do you agree with the following statements?

- **I try hard not to do things that will make other people avoid or reject me.**
- **I need to feel that there are people I can turn to in times of need.**
- **I want other people to accept me.**

- **I do not like being alone.**
- **I have a strong 'need to belong.'**
- **My feelings are easily hurt when I feel that others do not accept me.**

In how far do you agree with the following statements? (I do not agree at all/ I fully agree)

- I admire people who own expensive homes, cars, and clothes.
- Some of the most important achievements in life include acquiring material possessions.
- I like to own things that impress people.
- I enjoy spending money on things that are not practical.
- Buying things gives me a lot of pleasure.
- I often spend money on things I do not actually need just for fun of it.
- My life would be better if I owned certain things I do not have.
- I would be happier if I could afford to buy more things.
- It sometimes bothers me quite a bit that I cannot afford to buy all the things I'd like.

*I see myself as a person who is...*

- **curious.**
- **imaginative.**
- **artistic.**
- **widely interested.**
- **excitable.**
- **unconventional.**
- **playful.**
- **competitive.**

*Listed below are some characteristics that might describe a person: Caring, compassionate, fair, friendly, generous, helpful, hardworking, honest, kind. The person with these characteristics could be you or it could be someone else. For a moment, visualize in your mind the kind of person who has these characteristics. Imagine how that person would think, feel, and act. When you have a clear image of what this person would be like, please report in how far you would agree with the following statements.*

- **It would make me feel good to be a person who has these characteristics.**
- **Being a person who has these characteristics is an important part of who I am.**
- **I am actively involved in activities that communicate to others that I have these characteristics.**
- **I strongly desire to have these characteristics.**

*Now, we would like to ask you a few final questions. You support us a lot if you answer them.*

- **How old are you?**
- **Your gender?**
- **Your nationality?**
- **Your highest educational level?**
- **What is your family status?**
- **How high is your overall net household income per month, in other words the sum of all net incomes of the persons who live in your household?**
- **Do you work in an IT related business or study an IT related subject? Yes/ no**

*Everybody has hobbies. Nevertheless we would like to ask you not to click any of the fields to show us that you read the question text carefully. Sometimes it happens that participants are less careful at the end of a survey. We can check this by using such an attention check to get to know if the results are biased by less attentive participants. Thank you!*

- **to ride a bicycle**
- **fitness/ gymnastics**
- **hiking/ swimming/ running**
- **tennis/ squash/ badminton**
- **athletic sports**
- **football/ other ball sports**

*I am very certain of the aim of this study.*

*The aim of this study is (optional):*

*Now you have the possibility to give a personal feedback (optional):*

*Thank you for your support!*

# B. The Role of Incentives and Social Relationships

In Section 7.1, we discussed the reasons why device owners would participate in computational resource sharing systems. The analysis provides a high-level overview about possible factors that affect the willingness to contribute to computational resource sharing systems. However, it does not suggest any causal relationships nor does it provide any numerical evidence. Haas *et al.* suggest that incentives work differently for each type of social relationship [283]. Thus, in this section, we answer the following research question:

*How do different types of incentives and social relationships affect the willingness of device owners to contribute to computational resource sharing systems?*

To answer this question we conducted an experiment that builds upon our findings from the previous section. The experiment consists of an ex-ante survey ($n = 71$) to retrieve intended sharing behavior as well as a field experiment ($n = 8$) to investigate actual sharing behavior. Subsequent to the field experiment, we conducted an ex-post survey with the participants ($n = 5$). The experiments revealed insights on which type of incentive works how well for the different types of social relationships. Table B.1 shows the structure of the experiment with the relevant dimensions for incentives and social relationships. In the ex-ante survey, the incentive types *altruism* and *compensation* were tested for all types of relationships. In the field experiment, we tested all three types of incentives in combination with *scientific institutions*, *non-profit organizations*, and *for-profit organizations*.

## B.1. Experiment Setup

The experiment consists of two parts, the survey and the field experiment. In the ex-ante survey we asked respondents about their general attitude towards sharing and, more specifically, computational resource sharing. We further wanted to know with whom they would share their computational resources, what compensation they would expect, and what drives their decision to participate or not. In the field experiment, we asked subjects to install a mockup Tasklet client on their computers and to use this client to share their computational resources with three

| | | Incentive | | |
|---|---|---|---|---|
| | | Altruism | Gamification | Compensation |
| Social Relationship | Family | ▨ | | ▨ |
| | Friends | ▨ | | ▨ |
| | Anonymous users | ▨ | | ▨ |
| | Scientific institutions | ▨ ● | ● | ▨ ● |
| | Non-profit organizations | ▨ ● | ● | ▨ ● |
| | For-profit organizations | ▨ ● | ● | ▨ ● |

Table B.1.: Structure of the experiment. Combinations with a gray background were tested in the surveys. Combinations with a black circle were tested in the field experiment.

fictional computationally intensive projects. We monitored the sharing behavior over a one-month interval. To observe a realistic behavior subjects were made believe that the projects as well as the Tasklet client were real. The subjects were split up into incentive groups where each group was provided with different incentives for sharing resources. These groups were *altruism*, *gamification*, and *compensation*. Each group received a different briefing in the beginning of the experiment, motivating the respective kind of incentive.

Figure B.1 shows the timeline of the experiment. As a first step, we gave a briefing to each group and conducted the survey. Each group received a different treatment in terms of why attendants should participate in sharing their resources. The survey was conducted after the briefing. We invited the respondents to participate in the field experiment. Therefore, they had to download and install the mockup Tasklet client and run it over a one-month period to share resources with three fictional projects. These projects had been introduced in the briefing session and included a scientific project, a non-profit organization, and a for-profit organization. During the field experiment we monitored the sharing behavior. On the one side, we could measure how long each participant ran the Tasklet client and how much of this time the client was busy executing Tasklets. On the other side, we observed how participants configured the client in terms of how much and to which project they wanted to contribute. Further, we were able to analyze the clicking behavior. Following the field experiment, we conducted on

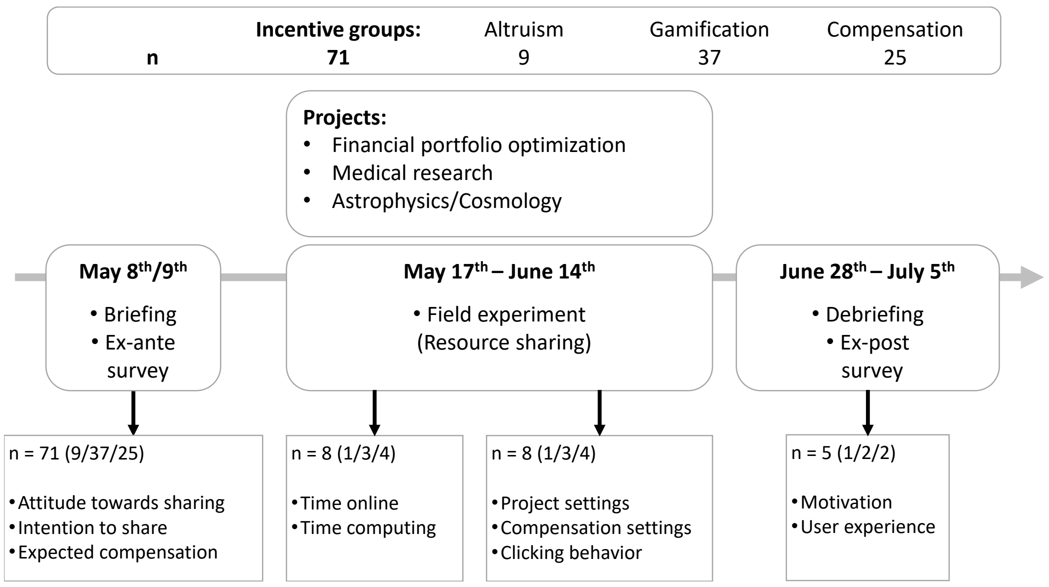


Figure B.1.: Timeline of the experiment. Subjects were divided into three groups and shared their resources for three projects. The lower part of the figure shows which data we obtain from each step of the experiment and how many subjects participate in each group (altruism/gamification/compensation).

online post-experiment survey to retrieve feedback about the user experience of the Tasklet client and the respondents' motivation to participate. We further performed the debriefing and informed the participants about the actual purpose of the experiment.

## B.2. Briefing

In the briefing session, we explained the overall idea of computational resource sharing systems and introduced the Tasklet system as one possible implementation of such a system. As people might be unaware of the existence of computational resource sharing and might wonder whether it is safe to share, we briefly discussed the idea of sandboxing which eliminates most of the threats in distributed processing [324]. Further, we ensured that all data would be treated confidentially and that the sharing behavior could not be attributed to a single person. We introduced the three projects that run computationally intensive programs and require additional computing capacities which they would access via the Tasklet system. These capacities should be provided by the participants in the field experiment. For each treatment group, we motivated the participation. We asked the attendants to fill in the survey and to agree in sharing their resources during the field experiment.

### Projects

We presented three projects to the attendants. Each project represented one category on the social relationship dimension. Even though all projects were fictional, we made the attendants believe that they would share their resources for real projects via the actual Tasklet system. This increased the perceived genuineness for the participants without causing any obligation towards real organizations or companies from our side. The three projects are presented in the following.

**Astrophysics/Cosmology:** This project is comparable to those that are typically deployed in the BOINC environment such as SETI@home [1]. The project represents a scientific institution as resource consumer which does not pursue any immediate monetary goals and makes the achievements available to the public.

**Medical Research:** The computational demand in medical research is enormous. Folding@home [268] is one out of multiple volunteer computing projects where complex biomedical computations are performed. Project owners are non-profit organizations that might include governmental health institutes that provide public goods but to not pursue monetary goals. The difference to the scientific project above is that medical research might be considered to have a more direct positive impact on society.

**Financial Portfolio Optimization:** This project represents a for-profit organization such as a large investment bank that uses complex algorithms to optimize the values of their portfolios. Contributing computational resources to this project means supporting this company in their pursuit of profit.

### Treatment Groups

We recruited subjects from three lectures where each lecture is taught in a different program. This ensures that the groups are disjoint and reduces the probability that members of different groups talk about the project and learn that each group has a different kind of incentive.

**Altruism:** Participants in the altruism group did not receive any compensation for their shared resources. Instead, they were told the multiple benefits of participating in sharing such as the support of organizations and the better utilization of computational resources. As additional computational load results in higher energy consumption and, thus, additional cost for the device owner, participants accepted to afford at least some money while sharing resources which can be compared to donating.

**Gamification:** In the gamification group, we encouraged attendants to participate by adding a competitive character to the resource sharing system. Participants earned credits for each successfully executed Tasklet and could compare their performance on a real time leaderboard. Therefore, they could compete against their fellow students and track who was the top contributor in the group. However, the credits that participants received did not have any monetary value and could not be traded for computation in return.

**Compensation:** Participants from the compensation group could earn money for each completed Tasklet. For each computational hour, they received a compensation of 0.11€. This compensation exceeded the additional energy costs which are estimated to be around 0.01€ and 0.03€ per hour. For each project, participants could charge a different price. The more the participants contributed, the more money they could earn.

## B.3. Survey

We asked the attendants in each of the three treatment groups to fill in the survey regardless of whether they wanted to participate in the field experiment. The survey contained 57 items that were grouped into 13 categories. Items regarding attitude were based on a seven-point Likert scale [325]. The scale ranged from 1 (*I do not agree at all/I would not share at all*) to 7 (*I fully agree/I would like to share)*. In some cases, the option *I do not know* was provided to allow for a neutral response. The items of the survey are summarized in the following.

**01) Participation in field experiment:** Respondents were invited to participate in the field experiment. In case of a positive response, they were asked to provide their email.

**02) Manipulation check:** To test whether the treatment worked as intended, we conducted a manipulation check. Respondents were asked to which extent they agree to three statements, each of which covered one of the incentives, i.e., altruism, gamification, and compensation.

**03) Attitude towards resource sharing:** Respondents were asked about their attitude towards resource sharing in general. Further, they evaluated the importance of the Tasklet system and whether they could make a positive contribution by participating in computational resource sharing.

**04) Sharing behavior:** Respondents were asked about their general sharing behavior in domains other than computation.

**05) Experience with computational resource sharing:** Respondents were asked about their general sharing behavior in domains other than computation.

**06) Attitude towards computational resource sharing:** This category included questions about the respondents' experience in information systems and their attidute towards compuational resource sharing. Further, we asked about their attitude towards playing games.

**07) Daily computer usage and general willingness to share:** Respondents were asked how many computers they owned and how long they use them on a daily basis. They also indicated their willingness to share the computational power of these devices.

**08) Willingness to share computation in absence of compensation:** Respondents rated their willingness to share their computational resources in different kinds of social relationships. No information about compensation was given to obtain information about the general attitude to share with resource consumers from these social relationship groups.

**09) Willingness to share computation for a compensation:** In the next step, we provided the opportunity for the respondents to select a compensation in return for sharing with consumers from different social relationship groups. For each group, respondents could share for free, ask for a compensation that covered the additional energy costs, ask for a compensation that made sharing profitable, or refuse to share at all.

**10) Reasons for sharing computation:** Respondents were asked to rate different reasons why they would participate in a computational resource sharing system.

**11) Interest in reciprocal computation sharing:** We explained the idea of reciprocal sharing where resource providers earn credits which they can trade in for additional computational resources once they run computationally intensive applications. We asked respondents whether they would be interested in participating in such a system.

**12) Obstacles for sharing computational resources:** Respondents rated potential obstacles which could prevent them from sharing their computational resources. We also provided an open text field in case a relevant obstacle was missing in the list.

**13) Demographics:** This category covered age, gender, and education.

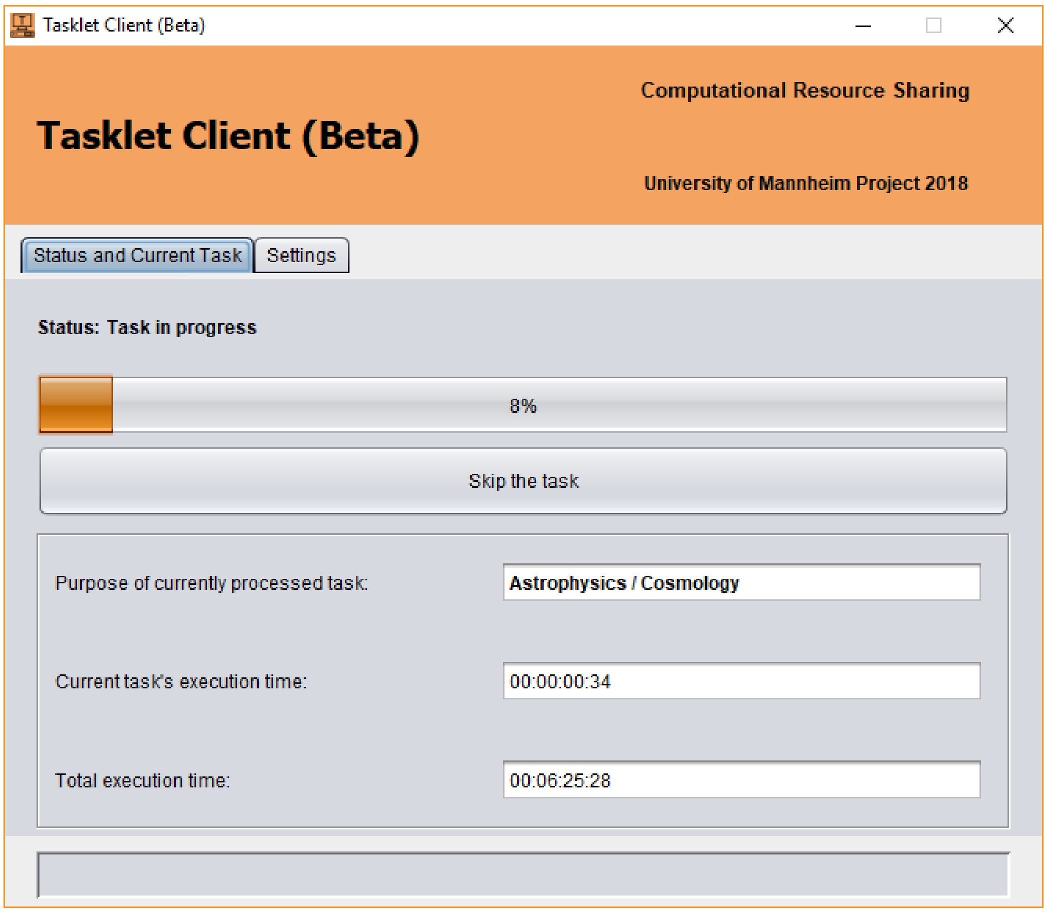
## B.4. Field Experiment

In contrast to the survey which gave us insights into the intentional sharing behavior, we designed a field experiment that allowed us to monitor real sharing behavior for the three treatment groups. For the experiment, we asked participants to install a Tasklet client which allowed to share their computational resources with three projects. Over a one-month period, participants could run the client on their computers. We ran a server that served as a central entity and tracked the sharing behavior of each user. Participants had to register with a unique identifier that allowed us to match the real sharing behavior with the survey data.

### Tasklet Client

The Tasklet client imitates the behavior of a resource sharing application. It is written in Java to ease the deployment on multiple platforms. When activated, the client sent periodic heartbeats to the server to allow monitoring the sharing behavior of each participant. As the clients were run on home computers and laptops that were typically located behind firewalls and routers that used network address translation, they were not directly addressable by the server.

Thus, each interaction was initiated by the client even though it seemed to the participants that the server offloaded Tasklets to them. In actual fact, each client created Tasklets at random times. To make the sharing system more realistic, we induced a random, meaningless workload which kept one core of the central processing unit busy. In this way, participants experienced the same effects as in



Status: Task in progress

8%

Purpose of currently processed task: Astrophysics / Cosmology

Current task's execution time: 00:00:00:34

Total execution time: 00:06:25:28

Figure B.2.: *Status and Current Task* tab of the Tasklet client in the altruism treatment group. The screen shows the progress of the current Tasklet execution as well as statistics about the current and previous executions.

an actual computational sharing system such as a slow-down of the device and an increased energy consumption.

The graphical user interface was developed in the style of the BOINC client application [9] and consisted of multiple tabs. For all treatment groups the tabs *Status and Current Task* and *Settings* were available. Figure B.2 shows the home tab of the Tasklet client. Users see the progress of the current Tasklet execution as well as the project of the current Tasklet, the passed execution time for this Tasklet, and overall execution time for the participant.

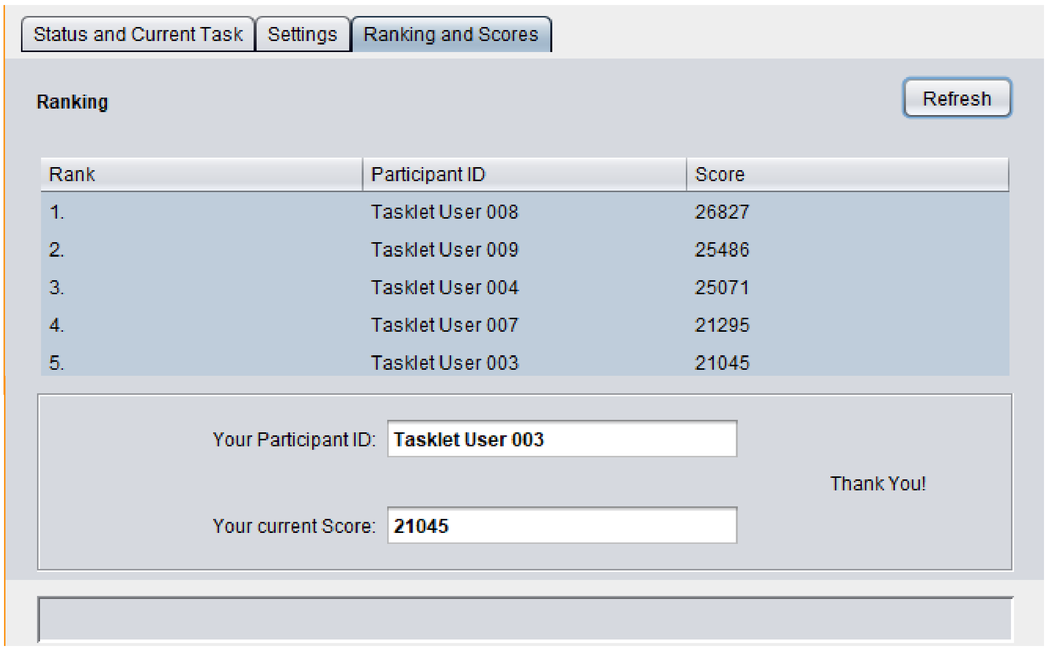
In the *Settings* tab, users had the chance to configure the workload of the client and select for which projects to share. A *low* workload resulted in $2-3$ Tasklet executions per hour, a medium and high workload to 4, respectively 6 Tasklet executions per hour. The execution time of Tasklets was uniformly distributed 6 and 8 minutes regardless of the performance of the computational device. Users could always stop the execution of the current Tasklet by clicking the *Skip the task* button.

Only participants from the gamification treatment group were provided with an additional *Ranking and Scores* tab that participants from other groups could not see. The tab showed the credits that the participant had earned by sharing resources (compare Figure B.3. In addition, a leaderboard ranked the performance of the user in comparison to other participants. The leaderboard was updated whenever the user opened the *Ranking and Scores* tab or clicked the *Refresh* button. To analyze the effect of this gamification element, we monitored the clicking behavior of the user.

The Tasklet client for participants in the compensation treatment group showed another tab. In this *Payment* tab, the users could set different prices for each of the projects. There were three price levels: .11€, .05€, and .00€ per hour of workload. Thus, participants could give discounts to one or multiple projects or even provide their resources for these projects for free. During the experiment phase, we collected usage statistics for all participants and logged each change in the settings.

### Post-Experiment Survey

The post experiment survey was conducted after the respondents had received a debriefing mail. The goals of this survey were threefold. First, we wanted to learn whether the attitude towards resource sharing and the Tasklet system in general had changed compared to the responses before the field experiment. Second, we asked respondents who did not install the Tasklet client about their reasons to understand how the setup of the client could be improved. Third, we asked the participants in the field experiment about the user experience of the Tasklet client based on the technology acceptance model by Viswanath Venkatesh [326]. We further wanted to know whether the participants had any concerns about privacy or trust [327].



Ranking

| Rank | Participant ID | Score |
|---|---|---|
| 1. | Tasklet User 008 | 26827 |
| 2. | Tasklet User 009 | 25486 |
| 3. | Tasklet User 004 | 25071 |
| 4. | Tasklet User 007 | 21295 |
| 5. | Tasklet User 003 | 21045 |

Your Participant ID: **Tasklet User 003**

Your current Score: **21045**

Thank You!

Figure B.3.: *Ranking and Scores* tab of the Tasklet client. Participants in the gamification group can track their performance and compare themselves to other participants.

## B.5. Results

This section presents the results of the experiment. We discuss the findings from the survey, the field experiment, as well as the post-experiment survey. As we conducted the experiment with only three groups of students, the amount of responses was limited. In total, 71 students responded to the survey, whereof 30 agreed to participate in the field experiment. Eventually, only seven subjects installed the Tasklet client and participated in the resource sharing and five responded to the ex-post survey. Table B.2 shows an overview over the participation in the experiment by treatment group and experimental stage.

| | Altruism | Gamification | Compensation |
|---|---|---|---|
| Answered ex-ante survey | 9 | 37 | 25 |
| Agreed to participate | 7 | 10 | 13 |
| Installed Tasklet client | 1 | 3 | 4 |
| Answered ex-post survey | 1 | 2 | 2 |

Table B.2.: Participation in the experiment by treatment group and experimental stage.

As the participation in the field experiment was low, a generalization of the observed sharing behavior is not possible. However, here, we present the observed

| | Altruism | Gamification | Compensation |
|---|---|---|---|
| *"I can donate my resources for a good cause."* | 5.56 | 4.70 | 4.73 |
| *"I can compete with my fellow students."* | 2.75 | 2.63 | 2.57 |
| *"I can receive a monetary reward."* | 2.25 | 3.11 | 5.14 |

Table B.3.: Manipulation check for the three treatment groups. The highest value for each row is highlighted.

behavior to provide an insight of which data has been collected and which analyses can be performed when the experiment is conducted in a larger scale which is not part of this thesis. Based on the results and the feedback from the participants we then give recommendations how to revise the experiment before running it with more participants.

### Results of the Ex-Ante Survey

From the 71 respondents in the ex-ante survey, 28% were female, 66% were male, and 6% did not specify. The average age was 22.2 years. 31 students pursued a Bachelor's degree (Altruism: 9/Gamification: 0/Compensation: 22) and 37 pursued a Master's degree (A: 0/G: 37/C: 25). Three students did not respond to this question. 35 students studied business administration or business economics (A: 0/G: 13/C: 22) and 32 studied business informatics (A: 9/G: 23/C: 0). 42% of the respondents agreed to participate in the field experiment (A: 78%/G: 27%/C: 52%). 11% actually participated (A: 11%$G$ :/8%/C: 16%). Hence, the highest intensions-behavior gap was observed in the altruism group, the lowest one in the gamification group. We performed a manipulation check to see whether the treatments had any effect. As each group got an individual briefing that highlighted either the aspect of sharing for a good cause, competing with fellows, or earning money, we expected to find these motivations in the responses for the manipulation check. The results in Table B.3 suggest that the treatment for altruism and compensation worked well. The gamification treatment, however, did not show any effect across the groups.

We asked respondents about their general attitude towards resource sharing. The results show that resource sharing in the altruism and compensation group was considered more important than in the gamification group and that participants in

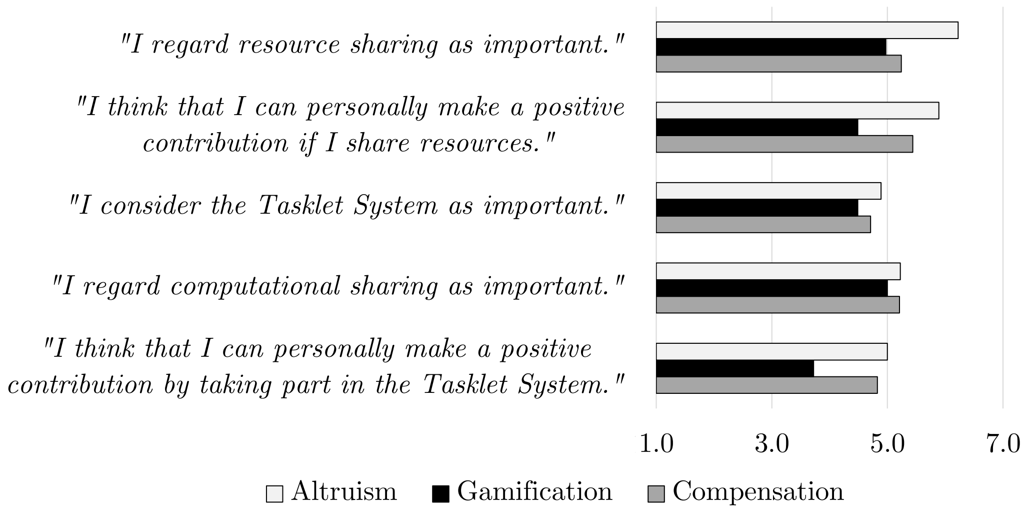


Figure B.4.: Reasons for sharing computational resources.

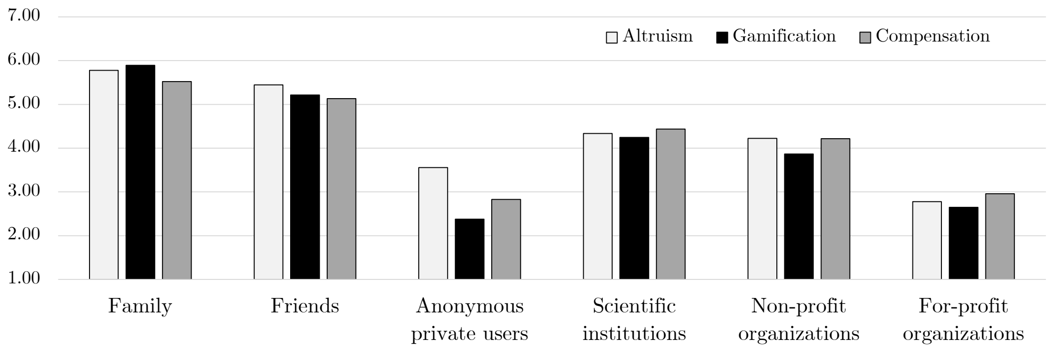


Figure B.5.: Willingness to share computational resources with different social groups.

the previous groups estimated their contributions as more helpful compared to the gamification group (see Figure B.4). For each of the five statements, the average acceptance was higher in the group of respondents who agreed to participate in the field experiment than for the other group who declined to participate.

Being asked about their knowledge and previous experience with computational resource sharing, almost half of the respondents (49%) stated that they were aware of the possibility to share their resources with others. However, only 10% had tried out computational resource sharing and only one respondent was sharing computational resources at that time. Five respondents (7%) were participating in mining virtual currencies such as Bitcoins.

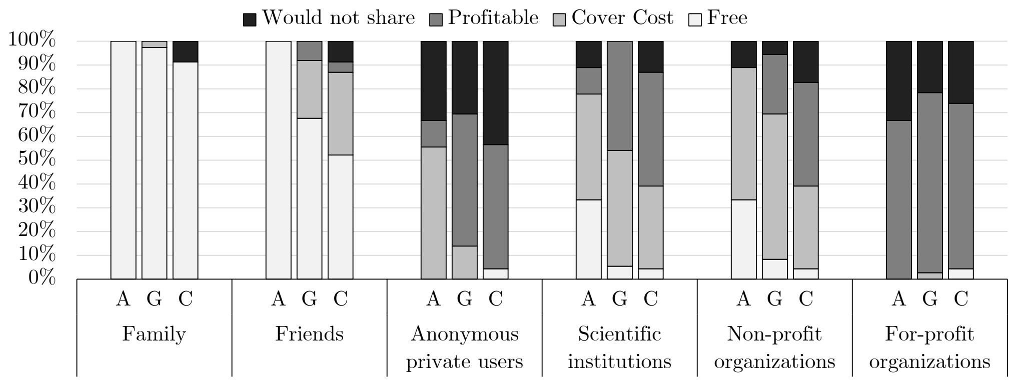


Figure B.6.: Responses to the question what compensation respondents would demand from resource consumers in a given social relationship. (A: Altruism ($n = 9$), G: Gamification ($n = 37$), C: Compensation ($n = 23$))

We wanted to know whether the social relationship to the resource consumer has an impact on the willingness to share. Therefore, we asked separately for each type of relationship how likely the respondents would share with consumers in each group. We did not mention any forms of compensation to obtain responses independent from what the participants would get in return. Figure B.5 shows that social relationships have a strong impact on the willingness to share resources. The results roughly identify three groups. The willingness to share is high for family and friends, medium for scientific institutions and non-profit organizations, and low for anonymous users and for-profit organizations. There are no major differences among the treatment groups.

In the next step, respondents were asked about the same groups of social relationships. However, this time, they could decide which compensation they would demand from each social group. They could also decide not to share at all. Figure B.6 shows the results of this set of questions. In 31% of the cases, resources would be shared for free. When compensation is involved, this number increases to 86%. Similar to the previous results in Figure B.5, the same three groups can be identified. The willingness to share for free or for a compensation that covers the additional energy costs is highest for family and friends and lowest for anonymous users and for-profit organizations.

The reasons for sharing were evaluated on a Likert-scale as well. In descending order and split up by the three treatment groups they were *using resources more efficiently* ($m_A = 5.44/m_G = 4.7/m_C = 5.26$), *contributing to scientific projects*

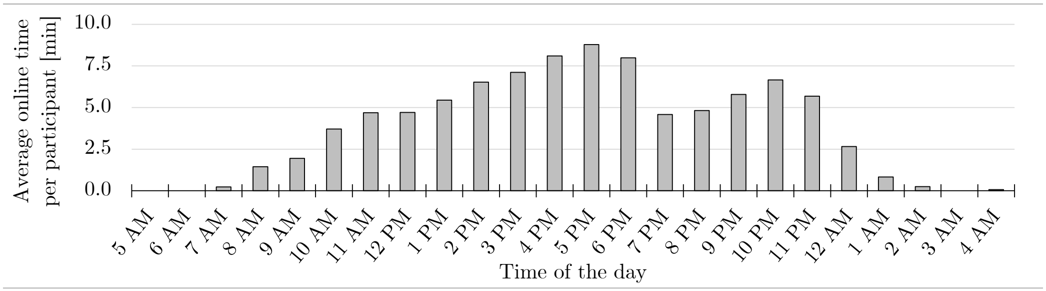


Figure B.7.: Sharing statistics by hour. The bars show the amount of minutes that on average were shared at a certain time of the day.

($m_A = 5.33/m_G = 4.46/m_C = 5.3$), *earning money* ($m_A = 3.33/m_G = 4.3/m_C = 4.87$), and *helping others* ($m_A = 4.33/m_G = 3.95/m_C = 4.52$). The interest in a tit-for-tat sharing system was at an average level ($m_A = 4.6/m_G = 3.86/m_C = 3.67$) indicating no strong demand for such a system.

Finally, when we asked about the reasons that could, in general, prevent respondents to participate in a computational resource sharing system, we identified three major reasons, namely *not wanting one's computer to slow down* (5.67), *being worried about data privacy* (5.5), and *being worried about security issues* (5.49). Less relevant reasons were *not seeing any benefits for oneself* (3.73), *not being aware of the possibility to participate* (3.45), and *considering the effort to install such a system as to high* (2.87). *Not having a desktop computer or laptop* did not play any role (1.18).

### Results of the Field Experiment

In the field experiment, we observed the usage behavior of the participants as well as their sharing settings over the course of the experiment. As only 8 students (1,3,4) participated in the experiment, we do not provide a detailed analysis of the three treatments here. However, the results should give an idea about which analyses can be performed when the experiment is repeated in a larger scale.

Due to the heartbeats that active Tasklet clients sent to the server in 5-second intervals, we could precisely measure how long each participant ran the client to share resources. Each heartbeat was logged which allowed us to retrieve the date and time of its arrival. We aggregated the data to learn about at which times the participants were running the client. In total, the clients were active for more than 300 hours and 1695 Tasklets were executed in almost 200 computing hours. Figure B.7 shows how many minutes a participant ran the Tasklet client on

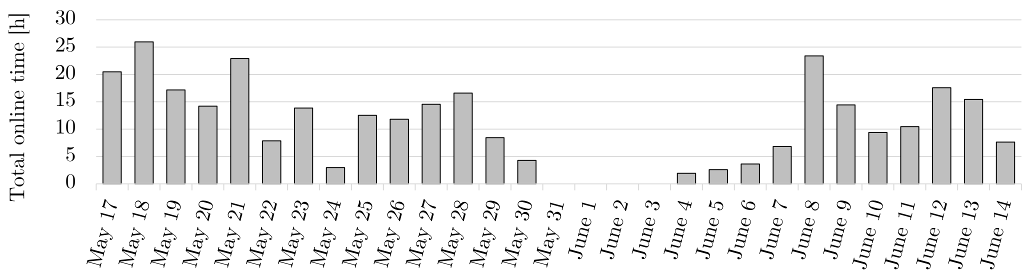


Figure B.8.: Sharing statistics by day. The bars show the total amount of hours which all participants together shared at that day.

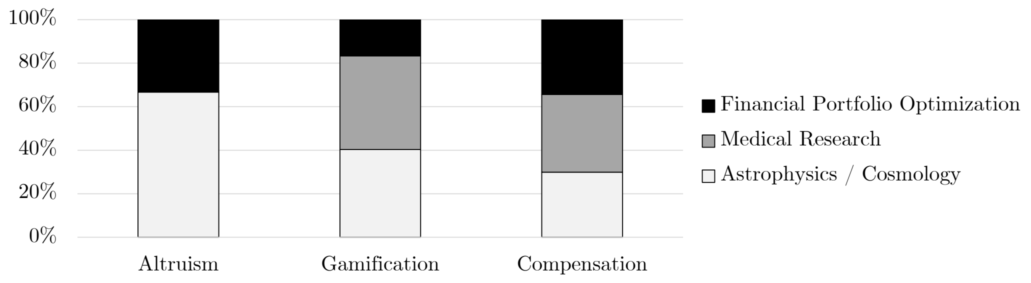


Figure B.9.: Participants could decide which projects they wanted to share their resources with.

average. The results indicate that participants did not leave their devices turned on at night but shared their resources distributed over the day with a tendency to share most in the afternoon. Figure B.8 shows the participation on a daily basis. The total amount of hours that all participants together shared their resources varied between less than 5 hours up to more than 25 hours. During May 31st and June 3rd, the server was offline and no data was collected. On May 22nd and June 8th, we sent out reminder mails to the participants which, in the second case, led to a notable increase in participation.

Participants could select projects for which they wanted to share their resources. This resulted in a different number of Tasklets executed for each project. Figure B.9 shows the proportions between executed Tasklets per project split up by the three treatment groups. Whereas participants in the *compensation* group did not show strong preferences for each of the projects, there was a discrimination of the financial portfolio optimization project in the gamification group. As there was only one participant in the *altruism* group, the results are highly prone to randomness and should be treated with caution.

| User ID | Treatment Group | Computation Hours | Active Hours | Tasklets Executed | Tasklets per Hour |
|---|---|---|---|---|---|
| **User01** | Compensation | 89 | 129 | 763 | 5.9 |
| **User02** | Gamification | 45 | 64 | 389 | 6.0 |
| **User03** | Gamification | 40 | 56 | 343 | 6.1 |
| **User04** | Compensation | 9 | 17 | 77 | 4.6 |
| **User05** | Compensation | 5 | 13 | 43 | 3.4 |
| **User06** | Compensation | 4 | 13 | 34 | 2.6 |
| **User07** | Gamification | 4 | 12 | 34 | 3.0 |
| **User08** | Altruism | 1 | 3 | 12 | 4.5 |

Table B.4.: Contribution per participant. Participants could set the workload, i.e., how many Tasklets they want to execute per hour.

Table B.4 shows the individual statistics for each participant. *User01* has the highest contribution and computed almost twice as much as *User02* who ranks second. Participants could set their workload to determine how many Tasklets should be executed per hour. The column *Tasklets per Hour* shows that the top contributors set the workload to *high* whereas the others selected a *medium* or *low* workload.

Participants in the *compensation* group could select the price for an hour of computation for each project separately. They could either request .11€ which would result in a profit, .05€, which is slightly more than the equivalent of the additional energy costs, or they could share their resource for free. The log files indicate that some participants in this group tested several prices but always came back to the highest price.

Participants in the *gamification* group could track their performance on a real-time leaderboard. To make the competition more engaging, we ran some Tasklet clients ourselves so that the leaderboard was updated frequently. To understand whether the leaderboard adds a gamification element to the Tasklet client and triggers a competitive behavior among the participants, we monitored how often a participant refreshed the leaderboard. The results show a high variance. Whereas *User02* and *User07* only refreshed the leaderboard 46 times, respectively 19 times, *User03* refreshed it 360 times over the course of the experiment.

**Results of the Ex-Post Survey**

Five participants responded to the ex-post survey. One respondent was from the *altruism* group and two for each the *gamification* and the *compensation* group. Even though the response rate was quite low, the ex-post survey provided some helpful insights into the usage of the Tasklet client.

The first block of questions was related to the user experience of the Tasklet client. In terms of its ease of use, the client received an overall high rating. The mean results over all five respondents were in descending order: *"Interacting with the Tasklet Client did not require a lot of my mental effort."* (6.4), *"I found the Tasklet Client to be easy to use."* (6.2), *"Installing and opening the Tasklet Client for the first time was easy."* (6), and *"My interaction with the Tasklet Client was clear and understandable."* (6). In terms of the perceived enjoyment, the client received an average rating: *"I found using the Tasklet Client to be enjoyable."* (4.2) and *"I had fun using the Tasklet Client."* (4).

There were no significant changes in the attitude towards resource sharing and the Tasklet system compared to the responses in the ex-ante survey. The results further show that the respondents trusted the Tasklet client (5.6 and 5.4) and that privacy issues were of no major concern (3.4 and 1.8).

Finally, all respondents stated that they would participate in a real-world system similar to the Tasklet system some day in the future. One respondent imposed the condition that the additional energy costs need to be compensated.

## B.6. Discussion

The experiment provided valuable insights into the attitude towards computational resource sharing. Further, it shed light on the link between incentives for computational resource sharing, social relationships, and the willingness to participate in a resource sharing system.

The results from the ex-ante survey confirm the intuitive model of Haas *et al.* [283] that states that incentives work differently for each type of social relationship. People do not only have a higher willingness to share resources with family and friends compared to anonymous users (compare Figure B.5) but also demand different kinds of compensations from resource consumers of different social relationships (compare Figure B.6). These results have two implications. First, a computational resource sharing system must allow participants to decide with whom they share their resources. In a system that hides the identities of resource consumers and resource providers, participants would be hesitant to share their resources. If the group of resource consumers could be narrowed down, the system would become usable also for those resource owners who would only agree to share with a certain group of people. Second, (monetary) compensation can make resource owners participate in resource sharing systems who were highly unlikely to share in the absence of incentives. Without incentives, 35 respondents of the ex-ante survey indicated that they were highly unlikely (1 or 2 on the Likert-scale) to share with anonymous users. After introducing incentives, only 24 respondents stated that they would not share in this context. This effect is stable over all social relationships. Thus, a computation resource sharing system must allow participants to discriminate prices for different groups of users.

Despite the small sample size of participants in the field experiment, some trends in the sharing behavior could be identified. The performance of participants in the *gamification* and the *compensation* group are similar (43 hours per participant versus 44 hours per participant, compare Table B.4). Given the success of projects based on the BOINC platform, which are only based on altruism and gamification incentives, it can be assumed that monetary compensation has an equally strong effect. Platforms like Bitcoins and Ether which attract a high number of contributors support this thesis.

The effect of gamification elements is yet to be determined. The fact that *User03* checked the leaderboard 360 times over the course of the experiment shows that gamification elements can trigger a competitive behavior which might result in a higher sharing performance. Thus, integrating gamification elements should be considered in the system design.

Against our expectations, no user let the client run over night to either earn more money or to gain more credits for the leaderboard even though this would be the times where the additional usage of the CPU would be least noticeable. However, participants seemed to share computational resources when their devices were turned on anyway.

Besides the insights into the role of social relationships and the incentives to share resources, there were further learnings regarding the field experiment. First, the Tasklet client was implemented as a Java program and, thus, required a Java virtual machine as runtime environment. Potential participants who had not installed this runtime environment might have considered the installation as tedious and therefore lost interest. Hence, an executable that directly runs on a platform might be the better choice. Possible alternatives could be a Java Executable Wrapper or a converter from a Java program to an executable. In the ex-ante survey, 30 respondents committed to the field experiment, only 8 respondents actually participated. As we did not get any responses from the 22 students who opted out, we cannot tell why they decided not to take part in the resource sharing. Thus, it would be useful to track who downloaded the Tasklet client which would give us more information at which step the respondents lost interest.

We also retrieved some feedback from participants to improve the Tasklet client. Several participants asked for an option that allows to set the priority of the Tasklet client to low in order not to impair the execution of user programs. Further, participants requested an option to run the Tasklet client in the background. Another request was related to closing the Tasklet client. An alternative to closing the client immediately would be to wait for the successful execution of the current Tasklet and then shut down the program.

# A. 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.
- Martin Pfannemüller, Markus Weckesser, Roland Kluge, Janick Edinger, Manisha Luthra, Robin Klose, Christian Becker, and Andy Schürr: CoalaViz: Supporting Traceability of Adaptation Decisions in Pervasive Communication Systems. In: Proceedings of the Second International Workshop on Mobile Ubiquitous Systems, Infrastructures, Communications and AppLications (MUSICAL 2019) in conjunction with the IEEE International Conference on Pervasive Computing and Communications (PerCom 2019), Kyoto, Japan, March, 2019
- Martin Pfannemüller, Janick Edinger, Markus Weckesser, Roland Kluge, Manisha Luthra, Robin Klose, Christian Becker, and Andy Schürr: Visualizing Adaptation Decisions in Pervasive Communication Systems. In: Proceedings of the 2019 IEEE International Conference on Pervasive Computing and Communications Demonstrations (PerCom Demos 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 (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 (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.
- Janick Edinger, Sebastian VanSyckel, Christian Krupitzer, Justin Mazzola Paluska, and Christian Becker: Developing a QoS-based Tasklet Trading System. In: Proceedings of the 6th International Workshop on Information Quality and Quality of Service for Pervasive Computing (IQ2S 2014) in

conjunction with the IEEE International Conference on Pervasive Computing and Communications (PerCom 2014), Budapest, Hungary, March, 2014

## B. Lebenslauf

| | |
|---|---|
| Seit 07/2013 | Akademischer Mitarbeiter<br>Lehrstuhl für Wirtschaftsinformatik II<br>Lehrstuhl für Enterprise Systems (seit 01/2019)<br>Universität Mannheim |
| 08/2010 – 06/2013 | Master of Science Wirtschaftsinformatik<br>Universität Mannheim |
| 08/2007 – 07/2010 | Bachelor of Science Betriebswirtschaftslehre<br>Universität Mannheim |