Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Hybrid Human-Machine Computing Systems

Provisioning, Monitoring, and Reliability Analysis

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Muhammad Zuhri Catur Candra
Registration Number 1028649

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Second advisor: Priv.Doz. Dr. Hong-Linh Truong

External reviewers:
Prof. Dr. Fabio Casati. University of Trento, Italy.
Prof. Dr. Harald Gall. University of Zurich, Switzerland.

Vienna, 28th April, 2016

Muhammad Zuhri Catur Candra

Schahram Dustdar

Declaration of Authorship

Muhammad Zuhri Catur Candra
Vienna, Austria

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 28th April, 2016

Muhammad Zuhri Catur Candra

Acknowledgements

All praise be to The Lord of the worlds, who has given us life, knowledge, and wisdom. My first and foremost gratitude goes to my parents, for always giving me sincere and unconditional supports, and to my family — my wife and my boys — who have made my journey cheerful and lively.

I would like to express my gratitude to my advisors, Univ.Prof. Dr. Schahram Dustdar and Priv.Doz. Dr. Hong-Linh Truong, for their guidance and supports to achieve this work. Also, I would like to thank all my colleagues at the Distributed System Group (DSG) for the fruitful discussions and collaboration, and especially to the DSG's secretaries, who always provide excellent supports.

Likewise, I am very thankful to the member of the Vienna PhD School of Informatics, especially Prof. Hannes Werthner, Prof. Hans Tompits, and Ms. Clarissa Schmid, who always assist me with any study-related issues and even many more, and to the students of the PhD School, for their sharing and caring.

My special thanks are devoted to my colleagues at the Knowledge and Software Engineering (KSE) Group, Bandung Institute of Technology, Indonesia, who have given me supports and sincerely let me off for my duty to embark on this long journey; and to my friends from Indonesia, especially at the Wapena club, who have made our life joyful and meaningful in this wonderful city of Vienna.

Last but not least, I am grateful to have financial supports from the Vienna PhD School of Informatics and the EU FP7 SmartSociety project.

Abstract

Modern advances of computing systems allow humans to participate not only as service consumers but also as service providers, yielding the so-called human-based computation. In this paradigm, some computational steps to solve a problem can be outsourced to humans. Such an interweaving of humans and machines as compute units can be observed in various computing systems, such as collective intelligence systems, Process-Aware Information Systems (PAISs) with human tasks, and Cyber-Physical-Social Systems (CPSSs). Even with the multitude realizations of such systems — herein we refer to as *Hybrid Human-Machine Computing System (HCS)* — yet we still lack important building blocks to develop an HCS, where humans and machines are both considered as first class problem solvers from the ground up.

These building blocks should tackle issues arise from different phases of an HCS' lifecycle, i.e., pre-runtime, runtime, and post-runtime. Each phase introduces unique challenges, mainly due to the diversity of the involved compute units, which bring in different characteristics and behaviors that need to be taken into consideration. This thesis contributes to some important building blocks in managing HCSs' lifecycle: the *provisioning* of compute units, the *monitoring* of the running system, and the *reliability analysis* of the task executions.

Our first contribution deals with the quality-aware provisioning of a group of compute units, a so-called compute units collective, by discovering and composing compute units obtained from various sources either on-premise or in the Cloud. We propose a novel solution model for tackling the problem in the quality-aware provisioning of compute units collectives, and employ some heuristic techniques to solve the problem. Our approach allows service consumers to specify quality requirements, which contain constraints and optimization objectives with respect to functional capabilities and non-functional properties.

In our second contribution, we develop a monitoring framework for capturing and analyzing runtime metrics occurring on various facets of HCSs. This framework is developed based on metric models, which deals with diverse compute units. Our approach also utilizes Quality of Data (QoD) to enable elastic monitoring catering different monitoring needs.

While the reliability analysis for machine-based compute units has been widely developed, the reliability analysis for HCSs has not been extensively studied. In our final contribution, we present models and a framework for analyzing the reliability of compute units collectives.

Contents

Abstract **vii**

Contents **ix**

List of Figures **xi**

List of Tables **xii**

List of Algorithms **xiii**

1 Introduction **1**
1.1 Overview 1
1.2 Motivating Scenario 2
1.3 Research Problems 4
1.4 Contributions 6
1.5 Scopes of Work 8
1.6 Thesis Structure 9

2 State of The Art **11**
2.1 Hybrid Human-Machine Computing Systems 11
2.2 Related Work in Provisioning of Compute Units 18
2.3 Related Work in Monitoring Framework 19
2.4 Related Work in Reliability Analysis 20
2.5 Chapter Summary 21

3 Models **23**
3.1 Architectural View 23
3.2 Model of Compute Units Collectives 26
3.3 Task Model 30
3.4 Chapter Summary 36

4 Runtime and Analytics Platform for Hybrid Computing Systems **37**
4.1 Prototype Architecture 37
4.2 Prototype Features 40

4.3 Chapter Summary 42

5 Provisioning **45**
5.1 Introduction 45
5.2 Provisioning Framework 46
5.3 Quality-Aware Collective Formation Problem 46
5.4 Formation Algorithms 51
5.5 Runtime Re-Provisioning 53
5.6 Evaluation 54
5.7 Chapter Summary 59

6 Monitoring **61**
6.1 Introduction 61
6.2 Metrics and Quality of Data 62
6.3 Distributed Monitoring Framework 69
6.4 Reasoning for Adaptation 71
6.5 Evaluation 73
6.6 Chapter Summary 78

7 Reliability Analysis **81**
7.1 Introduction 81
7.2 Reliability Models 82
7.3 Reliability Analysis Framework 83
7.4 Evaluation 88
7.5 Chapter Summary 95

8 Conclusions and Future Work **97**
8.1 Summary 97
8.2 Research Questions Revisited 98
8.3 Future Work 100

A Prototype Documentation **103**
A.1 Getting Started 103
A.2 Simulation Mode 105
A.3 Interactive Mode 110

Bibliography **115**

Glossary **129**

List of Figures

1.1 Infrastructure Maintenance Scenario 3

3.1 A System View on Coordinated HCSs 24
3.2 Conceptual HCS Runtime Architecture 25
3.3 Compute units collective Provisioning Overview 27
3.4 Task Meta Model 31
3.5 Collective Dependency 35

4.1 Prototype Architecture for Interactive Mode 39
4.2 Prototype Architecture for Simulation Mode 40

5.1 Compute Units Collective Provisioning Framework 47
5.2 An Example of Fuzzy Grade of Membership Functions 48
5.3 Construction Graph for Collective Formation Problem 50
5.4 Pruned Construction Graph for Re-provisioning 55
5.5 Sensitivity on objective weightings 57
5.6 Influence of α and β 58
5.7 Comparison on results of Ant Colony Optimization algorithm (ACO) variants 58

6.1 An Example of State Transitions for Human-based Tasks 64
6.2 Quality of Data in HCS Monitoring 67
6.3 Monitoring Framework 70
6.4 Monitoring Experiments Setup 73
6.5 An Example of Processing and Correlating Streams 74
6.6 Correlated Utilization Metrics 76
6.7 Quality of Data (QoD) Experiments 78
6.8 Number of Messages in Quality-Aware Delivery 79
6.9 Number of Messages in Varying Data Rates 79

7.1 VSU's Structure 85
7.2 Collective Dependency for Reliability Structure 85
7.3 Reliability on task executions, $R(k)$ 91
7.4 Reliability on varying size of resources pools 93
7.5 Reliability on different compute units collective provisioning strategies 94

List of Tables

2.1 Examples of Existing Hybrid Human-Machine Computing Systems 17

3.1 An example of a task specification 33

5.1 Formation algorithms' results and performance comparison 56

6.1 Metric Examples 63

7.1 Reliability Analysis Experiment Scenarios 91
7.2 Compute units collective cost and response times 95

List of Algorithms

5.1 Ant-based Solver Algorithm 54

6.1 Algoritm for QoD-Aware Data Delivery 68

7.1 EST Generation Algorithm 89

CHAPTER 1

Introduction

1.1 Overview

Humans traditionally employ computers to solve computational problems. However, despite of significant advancements of computing technologies in the past decades, computers still do not posses the basic conceptual intelligence that most humans take for granted [1]. In this regard, human-based computation paradigm emerges to harness the capability of human brains. In this paradigm, some computational steps to solve a problem can be outsourced to humans, i.e., the computer asks humans to solve problems, then collects, interprets, and integrates the solutions [2].

For this approach to work, typically so-called human tasks are modeled, instantiated, and distributed to humans according to the problem domain. In this context, humans no longer become merely computing service consumers. Instead, together with machines, humans are also *compute units*, i.e., resources providing services capable of processing input data into a more useful information in a (semi-)automated manner.

Examples of such human tasks include data processing tasks (e.g., collection [3], filtering [4, 5], classification [6, 7], verification [8]), query answering (e.g., [9, 10]), prediction (e.g., [11, 12]), and artifacts creation (e.g., [13, 14]). Furthermore, we have also seen various sources of online human workers, such as crowdsourcing marketplaces, e.g., [15, 16, 17], and social networks, e.g., [18, 19, 20, 21] being utilized as compute units. Also, we have been witnessing the development of special-purposed collective systems, which are utilized for harnessing human computation capabilities to work together with software units, e.g., [22, 23]. These so-called *collective intelligence* systems [24] intertwine humans and machines in Internet-scale, both as active problem solvers.

Moreover, advances in *Cyber-Physical-Systems* (CPSs), e.g., [25, 26], have enabled an interconnection of the physical-world, which consists of smart-things in embedded systems, and the cyber-world, which provides software-based computations. A natural advancement of CPS in today highly connected social-world is the inclusion of human actors, hence leading to the advent of *Cyber-Physical-Social Systems* (CPSSs), e.g., [27, 28].

We have also seen the manifestation of the human-machine integration in other systems. For example, *Process-Aware Information Systems* (PAISs) with human tasks integrate human-based and software-based services into processes, e.g., [29, 30]. Further innovations, such as [31, 32], allow even more advance integration of humans as socially-connected actors into business processes.

We refer to those systems as *Hybrid Human-Machine Computing Systems (HCSs)*, i.e., systems employing humans and machines as compute units, where tasks are distributed to humans and machines, and solutions from both humans and machines are collected, interpreted and integrated. An HCS manages diverse types of compute units. In such systems, we deal with the interweaving of human actors in the social-world, which we call *human-based compute units* (e.g., people in social networks, and in collaboration platforms), and *machine-based compute units*, which consist of compute units in the cyber-world (e.g., software, cloud-based services, etc., namely *software-based compute units*) and in the physical-world (e.g., sensors, actuators, gateways, etc., which we refer to as *thing-based compute units*) [33].

In a hybrid human-machine collaboration, a group consisting of diverse compute units is composed to execute a task requiring collaboration of humans and machines given by a consumer process or application. We refer to such a group as *a compute units collective*, which constitutes a construct for a flexible group of human-based and machine-based compute units, which can be composed, deployed, executed, and dismissed on-demand [34].

Despite of the myriad materializations of hybrid human-machine computation, yet we still lack important building blocks to develop an HCS, where both, machines, and humans are considered as first class citizens from the ground up. For example, crowdsource-based applications typically focus on rather simple tasks with minimum interactions [35]. Moreover, in typical collective intelligence systems, existing quality control approaches traditionally rely on primitives and hard-wired techniques, which do not allow consumers to customize the provisioning of compute units collectives based on their specific requirements [36]. Meanwhile, the models of human interactions in business processes, e.g., [29], still focus on rather self-contained human tasks, where only relatively static role models are employed for selecting the right people [19]. Furthermore, although Internet-scale collective intelligence systems are already managing global collectives, typically they are able to support only coarse human-machine interactions, such as data search [22].

1.2 Motivating Scenario

To motivate the importance of realizing the proposed building blocks for HCSs, we discuss a scenario where an information system for infrastructure maintenance is utilized in smart buildings or smart cities. The system can be employed, for example, by a corporation for maintaining large building complexes, or by a government agency in a city management setting. The maintenance is conducted by analyzing a possible facility breakdown, as shown in Fig. 1.1. For this system to work, sensors are installed on the

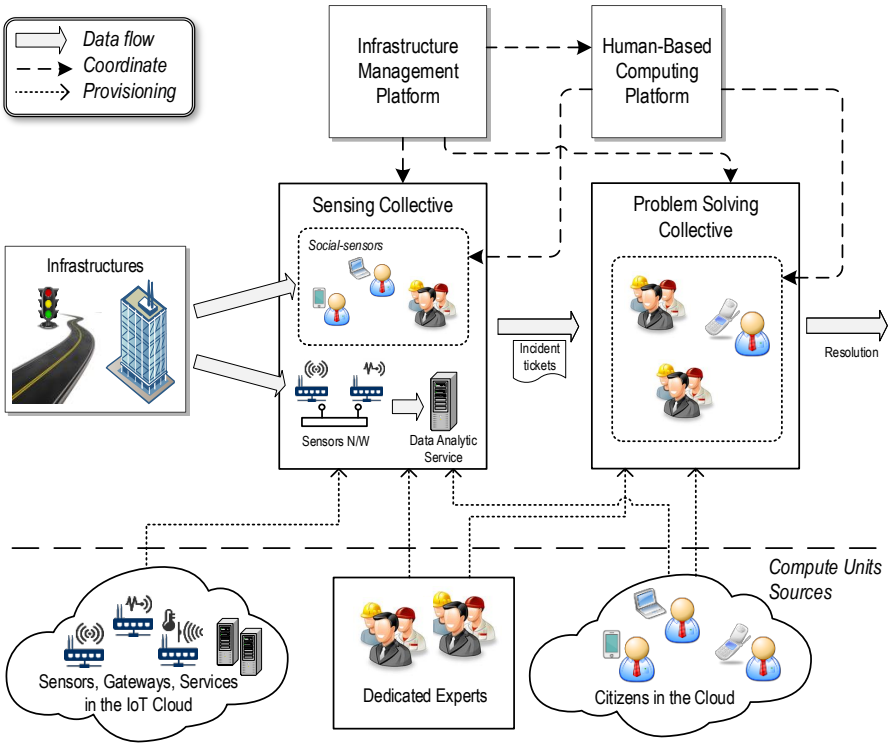


Figure 1.1: Infrastructure Maintenance Scenario

monitored facilities. These sensors capture events occurring on the facilities, which are streamed through sensor networks to a data processing center running a data analytic service.

In many cases, installing sensors on every facilities is not always feasible and adequate. One traditional way to handle this issue is to send dedicated experts for regular inspections. However, such approach can be in-effective for a large maintenance area. Therefore, citizens are also engaged to provide the so-called social-sensor services for revealing issues in places where hardware-based sensor systems or manual inspections are not feasible. For places where citizen participations are high, we may no longer need to employ experts for manual inspections. Hence, the provisioning of human-based compute units can be made on-demand.

These units, i.e., hardware-based sensor systems (i.e., sensors, gateways, sensor networks, and data analytic services), citizens as social sensors, and dedicated inspectors, can be seen as a collective performing sensing tasks for detecting facility breakdowns. When this sensing collective detects a (possible) breakdown, an incident ticket is generated, which then handled by a collective of compute units for problem solving.

To coordinate the human-based compute units, i.e., citizens and dedicated experts, a human-based computing platform, e.g., a crowdsourcing platform, can be employed. This platform generates human-based tasks, such as data collections and assessments, and

disseminates the tasks to the participating human-based compute units according to their availabilities and locations. The overall process flow is coordinated by the infrastructure maintenance platform.

In this scenario, the compute units involved in the process can be provisioned from different sources. For example, the human-based compute units can be provisioned from dedicated sources of experts or from crowdsourcing marketplaces, and the compute units for the data analytic service can be provisioned from the cloud.

1.3 Research Problems

Many problems encountered in the domain of hybrid human-machine computing arise from the inherent fact that an HCS comprises diverse compute units, where each of them introduces different characteristics, such as, qualities, and capabilities. Our work in this thesis addresses some important issues in building an HCS, which considers different characteristics of diverse compute units and various phases of an HCS's lifecycle as discussed in the following.

Each HCS may have different lifecycles depending on the problem domain and the underlying compute units platform. In general, for executing tasks in an HCS, we need to handle various facets of the system. For example, in the pre-runtime phase, we need to deal with the task design, and the provisioning of the compute units required to execute the task. In the runtime phase, issues such as task management, runtime monitoring, and adaptation, become concerns. While in post-runtime phase, issues, such as, quality analysis for future design improvements, are challenges that need to be dealt with. Our work presented in this thesis addresses some critical aspects of the HCS's lifecycle in pre-runtime, runtime, and post-runtime phases. Particularly we propose methodologies for the *provisioning* of compute units, the *monitoring* of the running compute units collectives, and the *reliability analysis* of the HCSs.

Our research performed within the context of this thesis specifically deals with the following research questions.

Research Question 1: *How can we provide a collective of diverse compute units for executing tasks in an HCS considering the consumer-defined requirements?*

Provisioning compute units collectives in HCSs brings in inherent problems due to the diversity of compute unit types involved. A compute units collectives provisioning approach should consider the quality requirements, which represent functional capabilities requirements as well as non-functional constraints. For example, in our infrastructure maintenance scenario discussed in Section 1.2, the consumer, e.g., a smart city government board, may specify that the social-sensing task should be executed by citizens with certain qualifications [23], e.g., no previous false report history, and properties, e.g., living and commuting in a particular location.

Traditionally, for machine-based compute units, consumers can specify precise requirements, such as minimum throughput, maximum delay, etc. However, for human-based

compute units, although we could measure some properties quite precisely, e.g., using key performance indicators [37], in practice, it can be more useful and practical to have a system that allows consumers to specify requirements in more flexible manner.

Compute units can be provisioned not only on-premise but also in the Cloud, such as cloud-based software services, or crowdsourced workers. For example, the pool of citizens as social-sensors in our infrastructure maintenance scenario can be provisioned from a crowdsourcing platform. Provisioning compute units in the Cloud introduces challenges, due to the huge search space for finding (semi-)optimal formation of compute units collectives. Heuristic techniques should be developed for dealing with such search space. Although various heuristics have been developed to deal with the Cloud-based compositional problem [38], they need to be adopted to be applicable in the context of HCS. Furthermore, to achieve a quality-aware provisioning, such heuristics must be guided with quality optimizing preferences defined by the consumers. Moreover, each provisioning strategies implemented using a particular algorithm may fit well for a particular situation or a problem domain. Hence, a provisioning framework should be made flexible so that it can support different strategies.

Research Question 2: *How can an HCS with diverse subsystems and diverse metrics models be effectively monitored?*

The diversity of compute units in HCSs also introduces challenges for monitoring such systems. For example, for monitoring the system in our infrastructure maintenance scenario, we have to deal with diverse compute units, such as the sensors and gateways in the physical-world, software-based data analytic services, as well as the people as social-sensors. Existing monitoring systems traditionally deal with homogeneous compute units, e.g., machine-based compute units monitor, such as software-based services monitoring systems (e.g., [39, 40]), and infrastructure/platform monitoring systems (e.g., [41]). An approach that takes such heterogeneity into account is required to effectively monitor HCSs.

An HCS consists of subsystems, e.g., human-based computing systems, and machine-based computing systems. Each subsystem in an HCS brings along various metrics, which could have corresponding metrics from other subsystems albeit having different definitions. For example, we could define the availability, utilization, and cost metrics for humans; however, their interpretation and measurement differ from the corresponding metrics for machine-based systems. To enable system-wide monitoring, we need models and methods to relate the corresponding metrics and bring them together as a unified metric.

Metrics from different subsystems of an HCS, although having related definitions, may have different qualities. Furthermore, different monitoring clients may require different qualities of monitoring data. The notion of Quality of Data (QoD), e.g., with respect to the accuracy and timeliness [42], plays a crucial role in a system where electronic data are ubiquitous. For example, a human-based client may prefer non-intrusive data (e.g., low data rate), while a software-based client may require much more frequent data.

Research Question 3: *How to measure the reliability of an HCS, which consists not only machine-based compute units but also human-based compute units?*

Reliability is an important measure of a system representing its resilience to operate as expected without unacceptable failures. Traditionally, the notion of reliability is expressed as a function in a continuous time space. However, for human-based computing, this approach is not suitable, since most human-based compute units do not operate continuously. For compute units collectives, where human-based compute units are involved, we need to model the reliability differently. For example, in our infrastructure maintenance scenario, we should be able to model the reliability of the citizen sensing capability in discrete time space, e.g., the probability of successful breakdown detection by citizens in 100 actual breakdown events.

The reliability of a system depends largely on the inter-dependencies between its elements. In the task executions within an HCS, the dependencies can be inferred explicitly from the process model, e.g., a workflow, if it is available. However, in many cases the collaboration inside a compute units collective can be more ad hoc. Hence, we need a model to describe the dependency in a compute units collective in a more agile and flexible way.

With the advent of the cloud computing, the provisioning of both, machine-based, and human-based compute units can be made on-demand from a virtually large pool of available compute units. In most cases, when a failure occurred on a running compute unit, another compute unit can be selected from the pool to replace the faulty one. The reliability analysis for cloud-based compute units collectives must take into account this provisioning model.

1.4 Contributions

The contributions of our work center around providing essential building blocks of HCSs. We focus on the following aspects of HCSs: (i) provisioning of compute units collectives, (ii) monitoring diverse compute units and task executions, and (iii) analysis of the reliability of the task executions. Firstly, we define models to abstract various HCSs with respect to the architectural views, compute units collective models, and task models, which are needed as conceptual foundations for building the main contributions. Furthermore, we develop a platform based on this abstraction, and prototype our proposed frameworks on this platform.

Our main contributions presented in this thesis address the aforementioned research questions and are described as follows.

Provisioning Collectives of Compute Units Our contribution in this part is to provide *a flexible quality-aware compute units collectives provisioning framework*, which provides a formation of machine-based compute units and human-based compute units obtained from diverse sources. This framework is flexible with respect to supporting different formation strategies. We develop a set of algorithms, especially a heuristic

based on the Ant Colony Optimization (ACO) approach, for dealing with the multiobjective compute units collectives formation problem. It is also quality-aware, since it takes the consumer-defined quality requirements into account. It deals with consumers' requirements for both, functional capabilities, and non-functional constraints. Moreover, it provides an approach to support strict and fuzzy quality requirements.

Monitoring Hybrid Human-Machine Computing Systems We contribute *a generic monitoring framework for HCSs*, which captures and processes events and metrics from diverse compute units. First, we present a metric model to handle metrics with different semantics, and utilize the notion of Quality-of-Data for enabling more effective and efficient monitoring in the context of diverse underlying systems. Based on those, we propose a distributed monitoring framework for HCSs.

Reliability Analysis In this part we develop *a framework for reliability analysis of compute units collectives*. We employ models to measure the reliability of individual machine-based and human-based compute units. To deal with the analysis of a large scale system, we introduce the notion of *virtual standby units*, which abstracts the group of compute units that are available for participating in task executions. Based on these models, we propose a framework for analyzing the reliability of task executions by compute units collectives.

The main contributions of this thesis have been published at acknowledged international conferences as follows:

1. *Provisioning Quality-aware Social Compute Units in the Cloud*,

   Muhammad Z.C. Candra, Hong-Linh Truong and Schahram Dustdar,

   The 11th International Conference on Service Oriented Computing (ICSOC 2013), December 2-5, 2013, Berlin, Germany [43].

2. *Analyzing Reliability in Hybrid Compute Units*,

   Muhammad Z.C. Candra, Hong-Linh Truong and Schahram Dustdar,

   The 1st IEEE International Conference on Collaboration and Internet Computing (CIC 2015), October 27 - October 30, 2015, Hangzhou, China [44].

3. *On Monitoring Cyber-Physical-Social Systems*,

   Muhammad Z.C. Candra, Hong-Linh Truong and Schahram Dustdar,

   (in submission).

Some work presented in this thesis are related to and have been partially funded by Smart Society project supported by the European Comission under the 7th Framework programme [1].

[1] `http://www.smart-society-project.eu/`

Additionally, the following publications are partially used in this thesis, although they are not directly related to the main contributions:

1. *Modeling Elasticity Trade-Offs in Adaptive Mixed Systems*,

   Muhammad Z.C. Candra, Hong-Linh Truong and Schahram Dustdar,

   The 22nd IEEE International WETICE Conference (WETICE 2013), Adaptive Computing (and Agents) for Enhanced Collaboration (ACEC) track, June 17-20 , 2013, Hammamet, Tunisia [45].

2. *Virtualizing Software and Human for Elastic Hybrid Services*,

   Muhammad Z.C. Candra, Rostyslav Zabolotnyi, Hong-Linh Truong, and Schahram Dustdar,

   Advance Web Services, (c)Springer-Verlag, 2012 [46].

1.5 Scopes of Work

Computing systems comprising humans and machines as compute units cover a wide spectrum with different characteristics. In Section 2.1, we discuss the landscape of the state of the art hybrid human-machine computing. Our work presented in this thesis focuses on HCSs with the following characteristics.

An HCS may consist of human-based compute units, software-based compute units, and/or thing-based compute units. Our work is motivated by issues raised due to the inclusion of human-based compute units in a computing system, such as discussed in Section 1.3. Therefore, this thesis focuses on systems where human-based compute units are involved. Systems focusing only on software-based compute units, and/or thing-based compute units, e.g., [47], although applicable, are less relevant in this thesis.

Our work assumes the existence of a coordinator role in the system, which manages tasks creation, distribution and execution in an automated manner. Depending on the problem domain, such a coordinator can be manifested in different forms, such as a process engine, an application middleware, etc. This excludes, for example, systems with ad-hoc interactions, e.g., for online collaborative content creation [13], and crowdsourcing systems where tasks are manually posted, and results are rather manually evaluated, e.g., [48].

Furthermore, our work necessitates the feasibility for applying a formal task formulation on the system, with respect to the multidimensional aspects of tasks, such as roles and activities that compute units could take, quality constraints, and the dependencies between task activities. Our main contributions presented in this thesis require such information for controlling quality-aware provisioning of compute units, monitoring constrained metrics, and analyzing task dependency structures for reliability analysis. Such a task formulation is discussed in Section 3.3.

1.6 Thesis Structure

The reminder of this thesis is organized as follows:

- *Chapter 2* analyzes the state of the art related to our work. In the chapter, we discuss the current landscape of hybrid human-machine computing, upon which we lay out the scope of our work. Afterwards, we present related work with respect to our main contributions, i.e., provisioning framework, monitoring framework, and reliability analysis.

- *Chapter 3* presents our architectural view on HCSs. This chapter also discusses the compute units collectives model and the task model, which are used throughout this thesis.

- *Chapter 4* describes the prototype implementation of our models, which provides a simulation testbed for evaluating HCSs' building blocks.

- *Chapter 5* focuses on our contribution in the domain of compute units provisioning. We present quality-aware provisioning strategies based on some heuristic formation algorithms. We evaluate our approach with respect to the comparison of different provisioning strategies, and we study the sensitivity of the approach with respect to different optimization objectives.

- *Chapter 6* provides details of our contribution for HCS monitoring. We present various metrics constructs to characterize HCSs and utilize the concept of Quality of Data (QoD) to deal with the dynamics of HCSs. Based on these models, we present a distributed monitoring and adaptation framework for HCSs. Furthermore, we perform experiments to showcase the benefits of our framework.

- *Chapter 7* discusses our contribution in the domain of reliability analysis for hybrid human-machine computing. We discuss reliability models for individual compute units and compute units collectives, and we present in detail the methodology to measure the reliability of task executions performed by the provisioned compute units collectives. Moreover, we exemplify our reliability analysis in a simulated experiments and show how the reliability analysis is useful for improving system's components.

- *Chapter 8* concludes this thesis and discusses future research directions in the context of hybrid human-machine computing.

CHAPTER 2

State of The Art

In this chapter, we discuss the present state of the art in the area of human-machine computing. Within the context of this area, we then discuss related work with respect to our main contributions presented in this thesis, i.e., provisioning approaches, monitoring frameworks, and reliability analysis techniques.

2.1 Hybrid Human-Machine Computing Systems

In Hybrid Human-Machine Computing Systems (HCSs), we deal with the interweaving of human-based compute units in the social-world, thing-based compute units in the physical-world, and software-based compute units in the cyber-world. In this section, first, we discuss techniques for virtualizing human-based and thing-based compute units, which enables the orchestration of these compute units together with software-based compute units. Afterwards, we discuss the landscape of existing systems consisting of diverse compute units.

2.1.1 Virtualizing Compute Units

Human capabilities have been incorporated into computing systems for solving complex problems since several years. Still, it is very challenging to program human capabilities due to limited techniques and tools [46]. One of the main challenges is to have similar level of pragammability for human-based compute units as we traditionally have with software. Traditionally, human computing platforms, such as *Amazon Mechanical Turk* [16] and *CrowdFlower* [49], expose the capabilities of human-based services via a set of platform-specific APIs. Such approach, albeit useful for programming human capabilities, still lack necessary abstractions for developing general-purpose mixed human-machine applications. Some techniques have been proposed to virtualize human capabilities for enabling a more seamless integration of human-based compute units into application. *AutoMan* [50] is an example of a human computing programming framework, which allows integrating

human capabilities into a standard programming language as ordinary function calls intermixed freely with traditional software-based functions. Another platform providing virtualization of human capabilities is *Jabberwocky* [51]. In Jabberwocky, machines and people are both first class citizens. Furthermore, it allows combining human-based compute units from different sources, and provides a high level domain-specific language for task declaring, which is translated to a map-reduce pattern [52].

The aforementioned virtualization techniques focus on the task requester side, i.e., defining task requests utilizing human capabilities, and do not concentrate on virtualization techniques on the service provider side, i.e., defining provided capabilities that can be discovered and provisioned on-demand. The needs and challenges of harnessing and orchestrating thousands of human brains, so-called *crowd services*, in real time using standardized Web service interfaces have been highlighted in [53]. An approach for describing and publishing *Human-Provided Services* (HPS) using traditional Web services is proposed in [54]. Furthermore, teams of people could also be established and provisioned under the service model, called *Social Compute Unit* (SCU) [55], which enables the unification of human-based and software-based services with the introduction of the virtualization layer [56].

In the physical-world, techniques for virtualizing capabilities of thing-based compute units have also been developed. Recently, the concept of *Sensing-and-Actuating-as-a-Service* (SAaaS) [57] and *Sensor-as-a-Service* (SenaaS) [58] emerge to allow the integration of sensors, actuators, and stream data processing capabilities into a service-oriented architecture. A framework called *Sensor Service Framework* (SSF) [59] is put forward to provide device-neutral features and APIs for the sensor devices to be deployed as Web services. In [60], a RESTful architecture is proposed to expose capabilities of smart things. An Internet-of-Things virtualization framework is also presented in [58] to support sensor event processing and reasoning for connected objects by providing a semantic overlay of the underlying IoT cloud. A platform called *servIoTicy* [61] is developed as part of the *COMPOSE* project [62], which provides a rich set of features to store and process data through Web services, allowing objects, services and humans to access the information produced by the physical-world connected to the platform.

We have provided detailed human-machine virtualization techniques in our previous work presented in [46]. Our work presented in this thesis contributes the important building blocks of HCSs consisting of the underlying subsystems (e.g., human-based and machine-based subsystems), which provide compute units virtualization mechanisms.

2.1.2 Systems with Diverse Units

The intertwining of physical, cyber, and social worlds has been evolved in the past decade from multiple directions. For example, the coupling of cyber and physical worlds has been seen in systems known as *Cyber-Physical Systems* (CPSs), which then further Internet-enabled by the advancement of *Internet of Things* (IoT). The next natural evolution of CPS and IoT is the *Cyber-Physical-Social Systems* (CPSSs), which integrate the existing social-world into the loop [63]. Furthermore, we have also seen the development of *collective intelligence* systems as manifestations of the cyber-social world, which enhances

computing systems by harnessing the computational power of humans [24]. Recent advancements in the cyber-social systems fuse the entities and characteristics of the physical-world into actions, e.g., spatiotemporal behavior and interactions with physical-world objects [64]. Meanwhile, recent developments of *Process-Aware Information Systems* (PAISs), also allow the integration of people activities and physical-world entities into process-based applications.

We discuss and exemplify such systems as follows.

Cyber-Physical-Social Systems

Cyber-Physical Systems (CPSs) consist of physical systems monitored, coordinated, controlled, and integrated by a computing and communication core [47]. A representative use-case of such CPS can be seen in a sport performance monitoring system, where athletes and their trainers collaborate and use measurement devices and supportive software-based analytic services to improve the performance of the athletes [25]. Other realizations of CPS are evident in many fields, such as in disaster and emergency managements, e.g., [26], healthcare, e.g., [65], and smart power grids, e.g., [66], to name a few.

With the dawn of the Internet of Things (IoT) technologies, such as IoT gateways (e.g., [67]), smart Things (e.g., [68, 69]), and software-defined IoT systems (e.g., [70]), it becomes possible to interconnect machines with smart embedded systems in Internet-scale, hence enabling the integration of the Things and the Cloud services.

Recent developments of CPS intensively include human actors in a socially connected world, which show the advent of Cyber-Physical-Social Systems (CPSSs). Hence, a CPSS is a system orchestrating three subsystems: (i) *the human-based systems*, i.e., the social system containing human actors and their interconnected devices/agents and/or social platforms, (ii) *the software-based systems*, i.e., the cyber-world providing software-based services including the underlying infrastructures and platforms, either on-premise or in the Cloud, and (iii) *the thing-based systems*, i.e., the physical-world that including sensors, actuators, gateways and the underlying infrastructures at the edge. Some examples of such systems can be found in smart home scenarios, e.g., [27], manufacturing systems, e.g., [28], and military command and control systems, e.g., [71].

Collective Intelligence

Collective intelligence has emerged as interconnected groups of people and computers, collectively doing intelligent things [72]. Collective intelligence represents a class of systems fusing cyber and social worlds. Furthermore, recent advancements of collective intelligence blends the physical world into the ecosystem, hence yielding systems with similar characteristics as in the aforementioned Cyber-Physical-Social Systems (CPSSs).

Notable examples of collective intelligence are the crowdsourcing platforms. According to [31], existing crowdsourcing scenarios can be categorized into three types:

i) *contest crowdsourcing* uses a contest to obtain the best available solution for a certain problem, such as in *99designs* [48] and *Threadless* [73],

ii) *task marketplace crowdsourcing* is a type of platforms in which typically simple and unrelated tasks are posted by clients, while registered workers will choose and solve the tasks, e.g., *CrowdFlower* [49], and *Amazon Mechanical Turk* [16], and

iii) *bid crowdsourcing* is a platform where complex problems submitted by clients and the best bid from professionals will be chosen to solve the problems, e.g., *InnoCentive* [74], and *TopCoder* [75].

Furthermore, frameworks, such as *Automan* [50], *Jabberwocky* [51], and *Turkomatic* [76], enhance task marketplace crowdsourcing platforms so that more complex tasks and workflows can be executed by the crowdsourcing workers.

Several crowdsourcing approaches focus on the enterprise environment. Some elaborated lists of research agendas for enterprise crowdsourcing are presented in [77] and [78]. The distinction between public and enterprise crowdsourcing is discussed in [79], especially what factors affect the sustainability of the project's community. Some examples of enterprise crowdsourcing solutions include *CrowdEngineering* [80] and *PeopleCloud* [81]. Moreover, some techniques have been proposed for utilizing social networks as workforce sources for enterprises, e.g., [18, 19, 82].

Other types of collective intelligence can also be identified according to their genomes [24], e.g., *what* gene (*create* or *decide*), *who* gene (*crowd* or *hierarchy*), and *how* gene (e.g., *collect*, *contest*, or *collaborate*). An example of a collective intelligence genome is a *collection* of artifacts *created* by a *crowd*, such as found in *Wikipedia* [13] and *DBpedia* [83].

Social sensor systems, where "humans as citizens in the ubiquitous Web acting as sensors and sharing their observations and views using mobile devices and Web 2.0 services" [84], are examples of collective intelligence in the physical world. Numerous research efforts have been done for achieving an effective social sensing in diverse problem domains [85]. Some examples of social sensor systems are real time collective disaster detection system, e.g., [86], citizen participation for data collection, selection, and assessment, e.g., *CrowdSC* [23], and mixed IoT-citizen sensing in smart cities [87].

Furthermore, hybrid collective adaptive systems, e.g., [22], are also utilized as new methodologies for solving complex problems that requires both human knowledge and machine capabilities. Moreover, some efforts have also been done to leverage online human collaboration technologies to enhance thing-based systems, e.g., [64].

Process-Aware Information Systems

A *Process-Aware Information System* (PAIS) is "a software system that manages and executes operational processes involving people, applications, and/or information sources on the basis of process models" [88]. Some examples of PAISs are workflow management systems (WfMS), process-aware groupware, enterprise information systems, etc.

In one of the de facto standards for business process modeling, *Web Services Business Process Execution Language* (BPEL) [30], people activities can be incorporated into processes using the *WS-BPEL extension for people* (BPEL4People) [29]. BPEL4People is based on *WS-HumanTask* [89], which defines the specification of human tasks, as well

as the programming interfaces and the protocol for advanced interaction with human tasks. Another standard for specifying business process is *Business Process Model and Notation* (BPMN). In BPMN 2.0 [90], human involvements in a process can be modeled using two different types of tasks, the *user task*, which is executed and managed by a business process engine, and the *manual task*, which represents an out-of-bound human task not managed by any business process engine.

Those constructions of traditional human tasks allow us to integrate people as compute units into processes. However, they still lack capabilities to exploit the computational power of online collective intelligence platforms. Firstly, they do not have a built-in support for discovering potentials assignees from non-organizational workforces. To this end, some approaches have been proposed to crowdsource tasks to collective intelligence platforms such as social networks, e.g., [19, 31, 91]. Secondly, these traditional process-oriented human tasks are typically executed by individuals. Other works, e.g., [32], extend traditional business processes to enable task executions by a group of socially-connected people. Moreover, traditional human task models for business processes do not allow the service providers, i.e., the humans who provide functionalities, to define and publish their capabilities as in typical service-oriented systems. To this end, some solutions are proposed, e.g., in [54, 53].

The integration of the physical-world into business processes has been gaining a lot of interest in the past few years. This integration is made possible by the service-enablement of smart things in IoT using various virtualization techniques as we have discussed previously, e.g., [59, 60, 58]. To model IoT capabilities within processes, new notational concepts need to be defined. A number of projects have been carried out to realize the so-called *IoT-aware processes* [92]. For example, several works such as [93, 94, 95, 96], have been proposed to model the integration of IoT entities into processes using well-known process modeling notations such as BPMN.

Such IoT-aware processes introduce novel challenges. To deal with IoT entities, a process needs to deal with unreliable data from unreliable compute units in a highly distributed manner. Furthermore, unlike traditional processes, which are typically deterministic, an IoT-aware process needs to be adaptive, where activities can be triggered based on detected events, or based on events generated by real time sensor data analysis [97].

2.1.3 Characterizing Existing Systems

To this point, we have discussed the present state of the art of systems engaging diverse units in physical, cyber, and social worlds. Such a landscape represents a very broad spectrum of diverse systems. To have a better understanding of the landscape of hybrid human-machine computing, here we characterize some systems based on several dimensions.

From the aforementioned works, we select 29 systems, which have not only proposed conceptual and design principles, but also running and evaluated systems. We characterize these systems according to the following dimensions:

i) *Diversity* indicates whether the system contains units from physical-, cyber-, or social-worlds.

ii) *Coordination* shows whether the system is coordinated by an automated software component, e.g., a middleware, to manage the creation, distribution, and execution of tasks.

iii) *Task Model* represents the characteristic of the system having a well-defined task model containing one or more of the following task aspects:

   a) *Task Request* indicates whether tasks are requested explicitly by the consumers (human or application consumers), or implicitly created, e.g., based on the occurring events, or voluntary human actions.

   b) *Role* shows the existence of the notion of role in the task model. This is typically relevant, when the system employs a task model containing multiple activities, each executed by a compute unit.

   c) *Quality Model* indicates whether the task design and execution contains a quality control mechanism, which can be defined by the consumer by the notion of quality constraints (e.g., a Service Level Agreement), or specified by the system designer, or manually monitored and imposed by the task requester (i.e., results are manually reviewed).

   d) *Activity Dependency* indicates whether activities inside a task have interdependency to each others.

The summary of the characteristics of the aforementioned systems is shown in Table 2.1. Here, we exclude systems that do not have a well-defined task model or a similar construct.

Our work presented in this thesis focuses on dealing novel issues due to the incorporation of people in the social-world as compute units. Hence, our work focuses on cyber-social and cyber-physical-social systems. Moreover, instead of presenting a new breed of systems, our work deals with characteristics found in existing systems to lay out foundations necessary to develop building blocks of hybrid human-machine computing systems.

The purpose of this system characterization is not to provide a comprehensive survey on hybrid human-machine computing. Instead, we seek to understand the relevance of our main contributions in the context of a larger landscape. Based on the scope of our work discussed in Section 1.5, we focus on systems, which (i) include humans as compute units, (ii) are coordinated in an automated manner, and (iii) have a well-defined task models consisting of roles, quality models, and/or activity dependencies.

Based on our scope of work, the examples of systems of our concern within the presented landscape of hybrid human-machine computing are represented in grey-shaded rows in Table 2.1. However, there are some exceptions where our main contributions are partially less relevant to some systems according to their characteristics. First, our reliability analysis framework presented in Chapter 7 depends largely on the activity

Category / System Title	*Refs*	*Diversity*			*Coordinated**	*Task Model*			
		Physical	*Cyber*	*Social*		*Task Request***	*Role*	*Quality Model****	*Activity Dependency*
Process-Aware Information System									
Processes with HumanTasks									
- BPEL4People, WS-HumanTask	[29, 89]		✔	✔	✔	Explicit	✔		✔
- BPMN 2.0	[90]		✔	✔	✔	Explicit	✔		✔
- PeopleCloud	[81]		✔	✔	✔	Explicit		consumer-driven	
- Social Compute Unit (SCU)	[32]		✔	✔	✔	Explicit	✔	consumer-driven	✔
- The Human-provided Services (HpS)	[54]		✔	✔	✔	Explicit		consumer-driven	
IoT-Aware Processes									
- Sensor Service Framework (SSF)	[59]	✔	✔		✔	Implicit	✔		✔
- WoT Architecture	[60]	✔	✔		✔	Implicit	✔		✔
- servIoTicy & COMPOSE	[61, 62]	✔	✔		✔	Implicit	✔		✔
- WSN extension for BPMN	[93]	✔	✔		✔	Explicit	✔		✔
- IoT and native services for BPMN	[94]	✔	✔		✔	Explicit	✔		✔
- BPMN-based WSN Application	[95]	✔	✔		✔	Implicit	✔		✔
Collective Intelligence									
Task Marketplace Crowdsourcing									
- Amazon Mechanical Turk	[16]		✔	✔	✔	Explicit		consumer-driven	
- CrowdFlower	[49]		✔	✔	✔	Explicit		consumer-driven	
Bid Crowdsourcing									
- Innocentive	[74]		✔	✔		Explicit		manual	
- TopCoder	[75]		✔	✔		Explicit		manual	
Contest Crowdsourcing									
- 99designs	[48]		✔	✔		Explicit		manual	
- Threadless	[73]		✔	✔		Explicit		manual	
Human Programming Platforms									
- Automan	[50]		✔	✔	✔	Explicit		system-driven	
- Jabberwocky	[51]		✔	✔	✔	Explicit	✔		✔
- Turkomatic	[76]		✔	✔	✔	Explicit	✔	consumer-driven	✔
Social Network as a Compute Unit									
- Crowdsourcing for BPEL4People	[19]		✔	✔	✔	Explicit	✔	consumer-driven	✔
- Tweetflow	[91]		✔	✔	✔	Explicit			
Artifact Creation Collaboration									
- Wikipedia	[13]		✔	✔		Implicit	✔	manual	
- DBpedia	[83]		✔	✔		Implicit	✔	manual	
Cyber-Physical-Social Systems									
CPSS Applications									
- CPSS for Smart Home	[27]	✔	✔	✔	✔	Implicit	✔	consumer-driven	✔
- CPSS for Production Network	[28]	✔	✔	✔	✔	Implicit	✔	consumer-driven	✔
- CPSS for Command and Control	[71]	✔	✔	✔	✔	Implicit	✔	system-driven	✔
Social Sensors									
- Smart City sensing using IoT & citizen	[87]	✔	✔	✔	✔	Implicit	✔	system-driven	
- CrowdSC	[23]		✔	✔	✔	Explicit	✔	system-driven	✔

Notes:

* Coordinated: there exists a coordinator role in the system (a software component), which manages tasks creation, distribution and/or execution in an automated manner.

** Task Request:

- Explicit: consumers (humans or applications) explicitly request tasks.
- Implicit: no explicit requests from consumers, e.g., event-driven tasks, or voluntary actions.

*** Quality Model:

- Consumer-driven: qualities are automatically controlled/enhanced based on consumer requirements
- System-driven: qualities are automatically controlled/enhanced based on pre-defined system objectives

Table 2.1: Examples of Existing Hybrid Human-Machine Computing Systems

dependencies during task execution. Hence, systems without the notion of activity dependencies are not relevant for our reliability analysis framework. Second, our provisioning framework discussed in Chapter 5 is a quality-aware provisioning framework, which honors the quality constraint specification. Although, the provisioning framework can still operate without a set of specified quality requirements, systems that do not have support for a quality model, or have only manual quality control, are not the focus of interest of our provisioning framework.

In the following sections, we discuss some research gaps in the current state of the art, and how our main contributions fill the gaps.

2.2 Related Work in Provisioning of Compute Units

Cloud-based Provisioning

In general, provisioning means the act or process of supplying or providing something [98]. In the context of Cloud-based computing, a provisioning process covers necessary activities to provide requisite compute units for performing a computation in an automatic or semi-automatic manner.

By that definition, a provisioning process for Cloud-based systems may encompass a broad range of activities, such as services provisioning covering the whole service lifecycle, including service construction, deployment, and operation (e.g., [99]), (virtual) machines provisioning (e.g., [100]), and storage provisioning (e.g., [101]).

In the context of software-based services, such as in workflow and mashup technologies, research on services composition has been going trough a long history with partially success stories [102]. Service composition comprises all processes required to provide added-value services from existing services [38]. These include, to name a few, notational modeling, interaction protocol, service level agreement and quality of services, the formation (i.e., selection) of compute units, deployment, and execution [38].

In IoT systems, provisioning activities include deploying IoT devices and the required artifacts (e.g., libraries) to run an application on the devices, e.g., [103]. *INOX* [104], a managed service platform for IoT, is proposed to provide IoT provisioning capabilities such as a flexible service deployment and a technique for virtualizing edge devices. Research on IoT discovery services has also gained interests. For example, in [105], a technique for discovering IoT services based on semantic matchmaking is proposed.

Techniques for managing a pool of compute units are domain-specific. In our work, the capabilities to discover, deploy, and managed the underlying compute units are abstracted into the so-called *compute unit manager* (see Section 3.1.2), which is utilized during the provisioning of compute units. Our contribution in the provisioning domain, as presented in Chapter 5, focuses to tackle issues related to the formation of compute units collectives, which incorporates both, machine-based compute units (i.e., software-based compute units and thing-based compute units), and human-based compute units.

Formation of Compute Units Collectives

A compute units collective is a group of compute units selected and deployed on demand for executing a requested task. Depending on the system and application domain, the formation of compute units collectives may contain a mix of human-based compute units, software-based compute units (e.g., software-based services), or thing-based compute units (e.g., IoT services). Many algorithms have been proposed for constraint-satisfying formation of compute units, especially in the context of Web service composition. Some of these algorithms employ certain heuristics to perform optimized formation, such as tabu search and simulated annealing [106], genetic algorithm [107], and integer programming [108].

In the context of IoT systems, several techniques have been proposed for discovering and selecting IoT devices or services, e.g., [109] and [110]. Unlike traditional service composition approaches, which usually focus on the problems of functional composition of services, IoT service platforms typically also focus on data processing scalability [61].

One of the contribution of our work is in the domain of human team formation optimization. In the context of this social-world, some approaches for team formation based on the fuzzy concept have been proposed, e.g., [111, 112]. Other works, such as [113, 114, 115, 116], also take the social network of the team member candidates into consideration.

Our work in the domain of formation of compute units collectives provides an optimized selection of compute units to perform tasks according to a set of specified requirements. Our work differs from the aforementioned works in the following aspects: (i) we combine the formation of human-based and machine-based compute units, (ii) we model optimization objectives in multiple dimensions, in particular, we exemplify our formation optimization in four dimensions: functional capabilities, connectedness, response time, and cost, (iii) we utilize the fuzzy concept to model the properties of compute units, and (iv) we employ some heuristics, especially Ant Colony optimization and greedy-based optimization, to form the compute units collective.

2.3 Related Work in Monitoring Framework

Monitoring Framework

Many techniques and tools for machine-based compute units have been developed for monitoring on various layers [117]. Monitoring tools on traditional distributed systems, i.e., grids and clusters, e.g., [118, 119], have been extended to cope with the Cloud characteristics [117], e.g., [120, 121].

Unlike the machine-based counterpart, there are not so many works carried out for monitoring the execution of human-based computing. The focus of existing research in this area is to develop quality improving techniques, which are typically domain-specific [36]. In PAIS with human tasks support, some tools, e.g., [122], are provided to monitor human tasks and their execution states, and to allow administrators to perform manual actions when necessary.

We position these related machine-based and human-based monitoring tools as the underlying interfaces for capturing events and metrics to be used in our proposed framework for a system-wide HCS monitoring, which allows creating more useful metrics by enabling correlation of diverse metrics from the underlying events and metrics.

Characterizing HCS

Metrics in software-based systems, including the underlying infrastructures, have been extensively studied, e.g., in [123, 124]. Although much less studied, metrics for people as computing units have also been proposed, e.g., in [36]. In thing-based systems, many works center around streams of metrics produced by the sensors (e.g., temperatures, locations, etc.). However, not many works have been published to define the metrics representing the qualities of the system itself. Some work, e.g., [125], propose metrics for improving the quality of the thing-based systems.

Quality of Data (QoD) plays a crucial role, especially in systems such as HCSs, where electronic data are ubiquitous. The majority of authors in the domain of QoD typically consider QoD from a basic set of quality dimensions: accuracy, completeness, consistency, and timeliness [126]. In our work, we apply the accuracy and timeliness (data rate and freshness) dimensions of QoD to enable more efficient data delivery in HCS monitoring.

The notion of composable metrics have also been proposed, e.g., in [127, 128]. These works focus on composable metrics in homogeneous systems. Our monitoring framework presented in this thesis deals with correlating and composing metrics from thing-based, software-based, and human-based systems.

2.4 Related Work in Reliability Analysis

Reliability of Human-Machine Computing

Quality controls have become one of the challenging obstacles in human-based computing, especially in the advent of crowdsourcing model [36]. Many approaches have been proposed to improve the reliability of human-based computing systems, especially with respect to the quality of results, e.g., [129, 130, 131]. These works deal with the reliability-improving approaches for simple tasks that can be assigned to individuals. Our reliability analysis framework focuses on more complex tasks executed by a collective of humans and machines, and how to measure the reliability of the task execution.

In the context of Process-Aware Information Systems (PAIS), where human tasks can be included, several techniques have been proposed to measure quality properties, especially reliability. In [132] and [133], the authors proposed a mathematical model to compute the quality of services by applying reduction rules to a workflow until an atomic task is obtained, and then the reliability is estimated based on formula $1 - (successful\ executions/scheduled\ executions)$. Hence, this value only provides an estimation of reliability for the next task execution, while our reliability analysis approach provides a mechanism to estimate the reliability in a discrete time space. Furthermore,

our framework allows reliability analysis for machine-based and human-based compute units obtained from a large pool of resources.

Several techniques for Human Reliability Analysis (HRA) have been developed in other disciplines such as safety and ergonomic engineering using a probabilistic model, e.g., [134], or using a cognitive theory, e.g., [135]. More advance approaches, e.g., [136], propose techniques to measure human performance reliability in real-time and on-line manner. Several works have also been conducted to formally model human behavior in computing systems, e.g., [137]. In our proposed reliability analysis framework, we adopt technique to measure the reliability of individual units on a task-basis using a failure rate parameter. This parameter can be obtained from these HRA techniques.

Reliability of Large Scale Systems

Research on the reliability of large scale systems, such as grid systems, e.g., [138, 139], and cloud services, e.g., [140, 141], has gained a lot of interest. These works proposed some models to analyze the reliability of hardware and software systems and proposed techniques to improve systems' fault tolerance. Some algorithms have also been proposed to solve non-trivial reliability equations as discussed in Section 7.3.2. However, to the best of our knowledge, currently there are no published works that provide models for the reliability analysis of hybrid human-machine systems as proposed in this thesis.

2.5 Chapter Summary

In this chapter, we presented the state of the art in the area of human-machine computing. First, we presented the existing underlying virtualization techniques enabling the computation by hybrid human-machine compute units. Afterwards, we discussed the landscape of existing hybrid human-machine computing systems, such as found in Cyber-Physical-Social Systems, Collective Intelligence Systems and Process-Aware Information Systems with human tasks. Upon this landscape, we laid out the scope of systems of our concern.

Furthermore, in this chapter we discussed some works related to the domain of our main contributions, i.e., existing techniques for provisioning of compute units, existing monitoring framework, and reliability analysis techniques.

CHAPTER 3

Models

Different manifestations of Hybrid Human-Machine Computing Systems (HCSs) have different characteristics and application models. In this chapter, we present our models on such systems with respect to the overall architectural view as well as the application model. Our models presented in this chapter lay out a foundational abstraction, which is used in the remainder of this thesis.

In this thesis, we focus on a class of HCSs employing a centralistic and task-oriented approach, i.e., systems that have a role of orchestrators to control and manage the distribution and execution of task requests. In the remainder of this thesis we simply refer to *HCS* as this class of such system. Such a class can be found in coordinated CPSSs, task-based collective intelligence systems, as well as in PAIS with human-based tasks, as discussed in Section 2.1.3.

3.1 Architectural View

In this section, we present our view on the coordinated and task-oriented HCSs, and present a conceptual runtime architecture containing necessary building blocks for such HCSs.

3.1.1 System View on Coordinated HCSs

We envisage HCSs as systems virtualizing the capability of humans, software, and things, as services [46]. As shown in Fig. 3.1, in such a system, an orchestrator coordinates the assignment and distribution of software-, human-, and thing-based tasks to the available and suitable services.

In such a system, while a consumer application is running, it instantiates tasks for fulfilling certain functions in the application. The tasks are then submitted to the orchestrator, so that the required compute units collective containing a mixed of human-based, and machine-based compute units can be provisioned to execute the tasks.

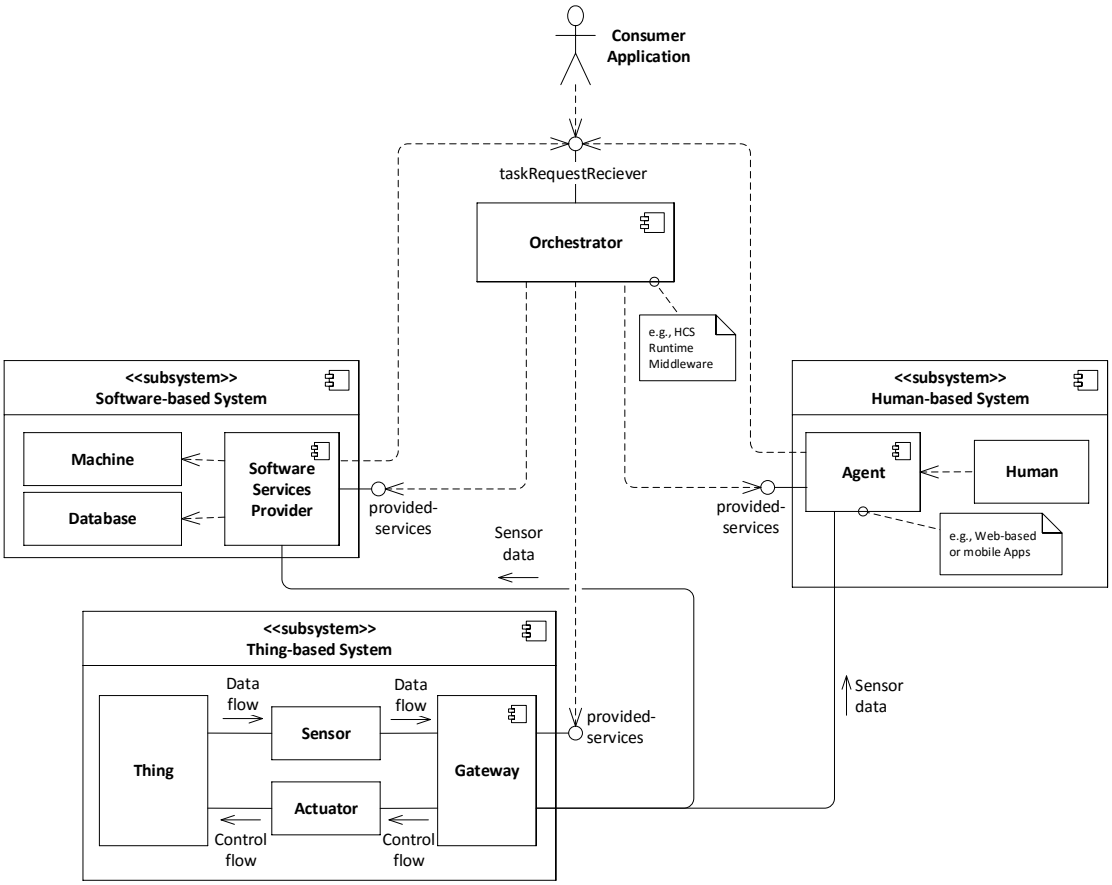


Figure 3.1: A System View on Coordinated HCSs

In a complex system containing thing-based subsystems, IoT gateways may also publish sensor data from the things through a sensor data bus (e.g., a messaging bus using CoAP, MQTT, XMPP, etc.) to software-based compute units, or to human-based compute units utilizing a human-friendly dashboard shown by a UI agent. In a typical scenario, during the execution of the tasks, a human- or software-based service may decide to make adjustments on the thing-based systems (e.g., sensor update rates) due to the incoming sensor data events. Such adjustment requests can then be translated (e.g., by an orchestrator) into another thing-based task and sent to the corresponding machine.

We discuss the architecture of HCS runtime middleware, which plays the role as an orchestrator in the following subsection.

3.1.2 Runtime Architecture of HCSs

Realizations of HCSs differ from one system to another. Here, we discuss a *conceptual runtime architecture*, which comprises necessary building blocks for coordinated and task-oriented HCSs as depicted in Fig. 3.2. This conceptual runtime architecture provides

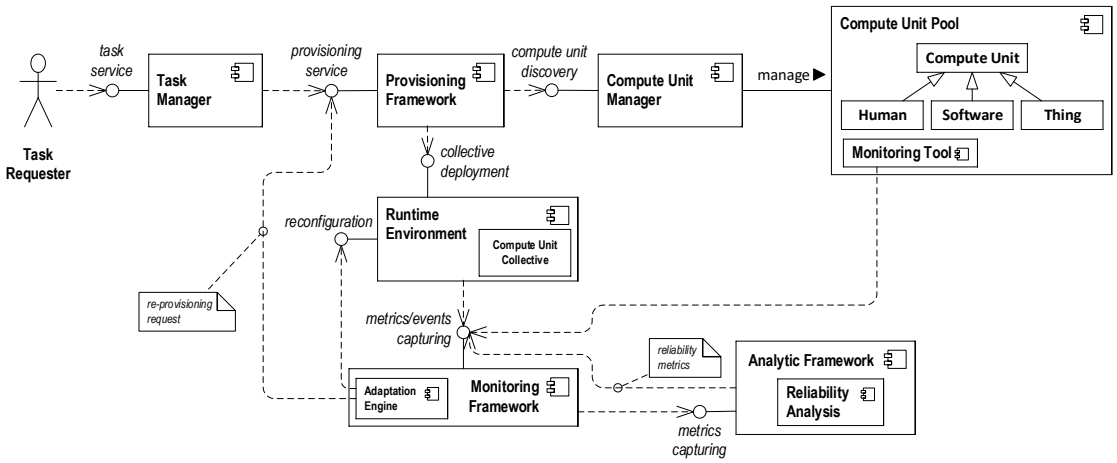


Figure 3.2: Conceptual HCS Runtime Architecture

an understanding on how an HCS runs. Moreover, this conceptual runtime architecture also portrays the position of our main contributions presented in this thesis in the overall hybrid human-machine computing ecosystem. Furthermore, we use this architecture as a foundation for our prototype implementation presented in Chapter 4.

In a task-oriented HCS, a runtime phase starts when a consumer (e.g., a client application or a human client) sends an instance of a task request, which is then scheduled for execution by a *task manager*. In a diverse system such as an HCS, the scheduling technique is domain-specific, and is beyond the scope of this thesis.

When a task is ready for execution, a provisioning request is sent to the *provisioning framework*, which then forms a compute units collective by selecting suitable compute units discovered by a *compute unit manager* (details of the provisioning framework is discussed in Chapter 5). Once the compute units collective is formed, it is then deployed to the corresponding *runtime environments*, where the assigned compute units can execute the task according to their roles (details of tasks and roles is in Section 3.3).

A *compute unit manager* is a component in an HCS capable of managing compute units, either in the Cloud or on-premise. A compute units manager manages the profiles of the compute units, e.g., their functional and non-functional properties. Examples of such a compute units manager are application-specific, e.g., crowdsourcing platform or a social network platform for human-based compute units, web service brokers for software-based compute units, or IoT Cloud managers for thing-based compute units.

One of main functionalities of a compute units manager is to provide a discovery service to find compute units suitable for a particular role for executing a task. More formally, given a set of all compute units $\mathcal{U}$, a set of requested capability functionalities, $\mathcal{C}p^r$, and a set of requested constraints, $\mathcal{C}^r$, a discovery service returns suitable compute units, $\mathcal{U}' = \{u_i | \forall u_i, u_i \in \mathcal{U} \wedge Sim(\mathcal{C}p^{u_i}, \mathcal{C}p^r) \wedge Comply(\mathcal{P}^{u_i}, \mathcal{C}^r)\}$, where $\mathcal{C}p^{u_i}$ is the set of capabilities owned by u_i, $\mathcal{P}^{u_i}$ is the non-functional properties of u_i, Sim is a predicate representing a similarity match, and $Comply$ is a compliance checking function according

to compliant operators defined for $\mathcal{C}^r$. An example of a capability similarity matching for human-based compute units is provided in [142]. For machine-based compute units such as software-based services, a semantic matchmaking based on the service description (i.e., input, behavior, and output) can be employed to decide the functionality matching [143].

Similar to the compute unit manager, *runtime environments* are also specific for different application models and different types of compute units. For example, it can be a collaboration platform (e.g., message-oriented such as email, or artifact-oriented such as online file collaboration platform), or a workflow engine with support of human tasks and sensor tasks, or a managed cyber-physical-social runtime platform.

During the lifetime of the tasks, the *monitoring framework* captures events and raw metrics from the monitoring interfaces of running compute units and from the runtime environments. These events and raw metrics may represent the behavior of a particular compute unit, a particular compute units collective, or a particular task execution, as well as the behavior of the overall system. The monitoring framework processes these events and raw metrics to produce requested metrics, which provide insights for the consumer to evaluate and improve system components. These metrics can also be used by an *adaptation engine* to reason about actions, e.g., to re-provision one or more failing compute unit or to reconfigure the runtime environments, which can be necessary to achieve correct functioning of the system. We detail the monitoring framework in Chapter 6.

During runtime or after runtime, we may need to perform some analytics to obtain deeper comprehension of the system. In this thesis, we focus on analytics to obtain the reliability metrics of the system, as one of the important quality metrics in HCSs. The metrics required for reliability measurement are obtained from the monitoring framework, and the resulting reliability metrics then can be fed back to the monitoring framework for further processing, e.g., for adaptation. This reliability analysis for HCSs is discussed in Chapter 7.

3.2 Model of Compute Units Collectives

We use the notion of *compute units collective* as an abstraction representing the composition of collaborative human-based and machine-based compute units that applications need to execute a complex computing task. In today's Internet-based computing landscape, such compute units collectives can be dynamically provisioned on-demand from the Cloud or on-premise. Some examples of such compute units pools include (i) for human-based compute units, task-based crowdsourcing platforms (e.g., in [15, 16, 17]), collections of experts on social networks (e.g., in [18, 19, 20, 21]), and enterprise compute units pools (e.g., in [144]), (ii) for software-based compute units, cloud-based software-based services provisioning (e.g., [143]), and (iii) for thing-based compute units, the emerging IoT Cloud systems (e.g., [68, 69, 70]).

In this section, we present a model describing the on-demand provisioning of compute units collectives, and we discuss some compute units' properties, which are necessary for the execution of the requested tasks.

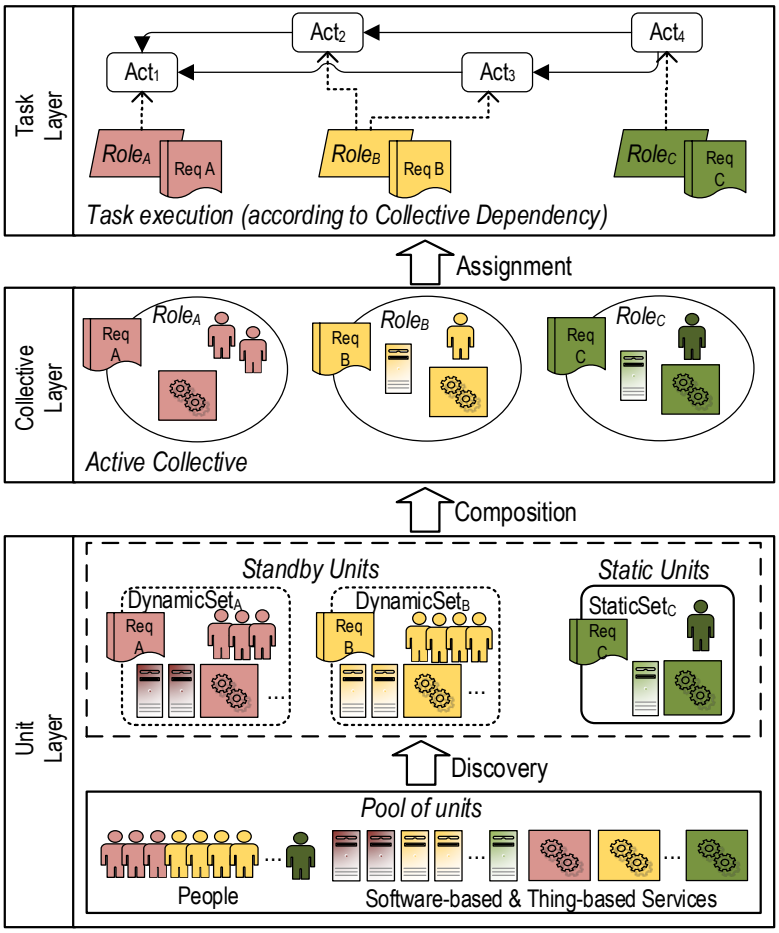


Figure 3.3: Compute units collective Provisioning Overview

3.2.1 Compute Units Collectives Provisioning Model

In task-oriented HCSs, we deal with compute units collectives provisioned on-demand to execute tasks. Here, we discuss a typical compute units collective provisioning model, as shown in Fig. 3.3.

The requested task contains a set of required roles, $Role_x$, which need to be fulfilled (see Section 3.3.1). Each roles execute certain activities for the task, Act_x. For each role, a set of requirements, Req_x, can be defined to guide the discovery and selection of the compute units (see Section 3.3.2), e.g., the requirements may contain a set of qualifications for discovering compute units. Qualified compute units are composed to form a compute units collective to fulfill each roles defined in the task request.

Compute units qualified to perform a particular role are discovered from diverse compute units pools. These discovered compute units may represent two types of compute units groups: (i) a static set (i.e., pre-defined) of compute units, e.g., in the case of in-house provisioning, or (ii) a dynamic set of standby compute units, which can be

assigned to a particular role on-demand, e.g., in the case of on-demand provisioning such as cloud-based services provisioning or crowdsourced human-based compute units provisioning.

3.2.2 Properties of Compute Units

Each compute unit has its own properties, which define the functional and non-functional characteristics of the compute unit. These properties are important during the provisioning of compute units collectives so that we could discover and select only compute units that meet requirement constraints. Furthermore, some properties of compute units may also dynamically change from time to time representing useful metrics that needs to be monitored.

Functional Capabilities

A functional capability (or a *capability* for short) of a compute unit represents its ability to perform certain function. It can be, for example, a functional service provided by a machine-based compute unit, e.g., a temperature sensing service, a software-based data analytic service, etc, or a certain skill provided by a human-based compute unit, e.g., a problem investigation skill, a content generation skill, etc.

A compute unit, u, has a set of capabilities, $\mathcal{C}p^u = \{(cp_1, x_1), (cp_2, x_2), ...\}$. The capability type, cp_i, defines the kind of function that the compute unit has or is endowed with, and x_i represent its capability level. There are two types of capability level, a *boolean* capability (i.e., it *has* or *does not have* the capability, in this case x_i always equals to 1, since we could simply omit non-existent capabilities) and a non-discrete capability level, e.g., $(0, 1]$, which defines a floating value representing the quality of the function. Non-discrete capability level x_i may be defined using different approaches. For example, it can be calculated based on a qualification test or based on a statistical measurement such as the acceptance rate [16].

Non-Functional Properties

A compute unit, u, may also has a set of non-functional properties, $\mathcal{P}^u = \{(p_1, x_1), (p_2, x_2), ...\}$. The property type p_i, defines the type of property that characterizes the unit such as performance, cost, location, etc. The value of the property x_i can be a static value, e.g., string or numeric values, but it can also be a function of a certain parameter, e.g., the estimated response time for a particular task.

Although our contributions presented in this thesis are flexible and extensible with respect to the reckoned compute units' properties, here we discuss some important properties, which are generally common for both human-based, and machine-based compute units, and are used to exemplify our provisioning framework discussed in Chapter 5.

Response Time For any compute unit u for executing a task t, a measured or estimated response time can be provided, i.e., $time : (u, t) \mapsto \mathbb{R}_{>0}$. An estimation of response time can be affected by the job queuing and assignment model, such as based on a maximum number of concurrent jobs (e.g., [16]), using a work queue approach commonly found in workflow management systems (e.g., [145]), or using a project scheduling approach considering the time availability of the candidates (e.g., [111]). The response time of a compute units collective $\mathcal{U}$ to execute a task t, i.e., $time(\mathcal{U}, t)$, can be defined as the time since the task t is requested until all the participating compute units in $\mathcal{U}$ have completed their roles in the task. Hence, it depends on the execution sequence of the task (see Section 3.3.4).

Cost Each compute unit u may specify its expected cost to perform a task t, which is modeled as a function of the task, $cost : (u, t) \mapsto \mathbb{R}_{>0}$. This function, for example, can be based on the estimated computing duration and the hourly cost. The cost for a compute units collective $\mathcal{U}$ to execute a task t can then be calculated based on the cost aggregation of its members, e.g., $cost(\mathcal{U}, t) = \sum_{u \in \mathcal{U}} cost(u, t)$.

Connectedness The success of a compute units collective also depends highly on the connectedness of its constituent compute units. For example, in the context of machine-based compute units, the network connectivity affects the performance, while in human-based compute units, the social relationship, e.g., whether they have worked together in the past, may also affect the performance of the group [144].

We define a connectedness graph as an ordered pair $G = (\mathcal{U}, \mathcal{E})$, where $\mathcal{U}$ represents a pool of available compute units, and $\mathcal{E}$ represents a set of weighted undirected edges between two different compute units in $\mathcal{U}$. We define an edge $e = \{u_1, u_2\} \in \mathcal{E}$ as an indication that u_1 and u_2 have certain relationship.

The type of relationship between compute units to be considered is domain specific. As an illustrative example, in the context of human-based compute units, the weight of the edge, $weight(e)$, can be defined as an integer number that represents the number of successful task completions, where both compute units have worked together, subtracted by the number of unsuccessful task completions. This weighting approach allows us to give penalty to, for example, malicious workers.

A compute units collective and its connectedness can be represented as a graph $G' = (\mathcal{U}', \mathcal{E}')$, where $\mathcal{U}' \subset \mathcal{U}$, $\mathcal{E}' \subset \mathcal{E}$, so that $\mathcal{E}'$ is the maximum subset of $\mathcal{E}$ that connects all compute units in $\mathcal{U}'$. Hence, for example, we can measure the connectedness of G' as the average weighted degree of all nodes:

$$conn(G') = \frac{\sum_{e \in \mathcal{E}'} 2 \cdot weight(e)}{|\mathcal{U}'|} \tag{3.1}$$

3.3 Task Model

Our work centers around tasks executed by compute units collectives. The task model used in our work is motivated by some requirements, which are necessary for developing our main contributions. More specifically, our provisioning framework (see Chapter 5), monitoring framework (see Chapter 6), and reliability analysis framework (see Chapter 7), require a task model with the following features:

- A task contains a set of (one or more) activities, which represent actions to be performed by compute units with particular goals or deliverables.
- For distinguishing the function of each compute unit while performing activities in the task, a set of roles should be defined and associated to each activities. The provisioning framework fulfill the roles for performing activities by selecting suitable compute units.
- A set of constraints, i.e., functional capabilities and non-functional requirements, is required to govern the selection of compute units to be included in the assigned compute units collective, and to monitor and control the quality of the task execution. Such constraints could be specified in either task-level, or activity-/role-level.
- Furthermore, the selection of compute units could also be controlled by a set of multi-dimensional objectives, e.g., cost-optimized vs time-optimized objectives, which can be used to optimize the provisioning of compute units.
- The reliability analysis of a compute units collective requires an information on the structure of the compute units collective with respect to the dependency of its constituent compute units. Hence, the task model should provide the dependency of compute units according to the activity they perform.

Hence, our task model is a 4-tuple, $t = (\mathcal{A}, \mathcal{R}, \mathcal{C}, \mathcal{O})$, where $\mathcal{A}$ is a set of activities, $\mathcal{R}$ is a set of roles and their requirements, $\mathcal{C}$ is a set of constraints representing requirements on the task-level, and $\mathcal{O}$ is a set of provisioning objectives. The structure of our task model is described in the meta-model shown in Fig. 3.4. We discuss the constructs of activities, roles, and constraints in the following subsections. The provisioning optimization objective is discussed in Section 5.3. Furthermore, we discuss the dependencies of activities in Section 3.3.4.

3.3.1 Task Structure

Tasks for different systems may have different structures and granularities. For example, in a typical service-based composition, e.g., [29, 143], or in a microtask crowdsourcing platform, e.g., [16], a task corresponds to a single activity executed by one compute unit. However, in a collaborative landscape, e.g. [144, 14], a complex task may contain multiple activities executed by multiple compute units in a collaborative work.

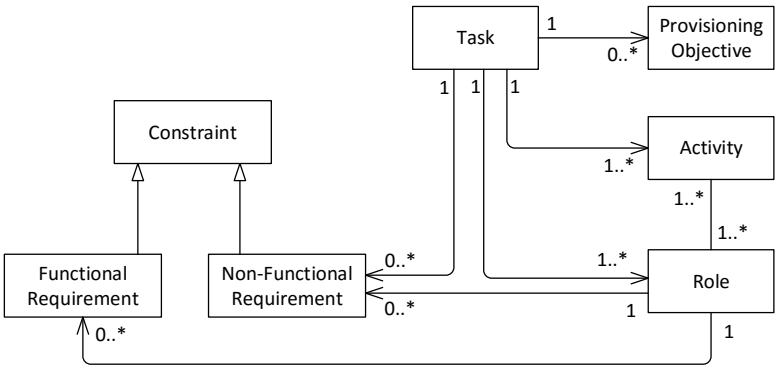


Figure 3.4: Task Meta Model

Here, we employ a role-based task model, where we could define a set of roles, $\mathcal{R}$, for executing a task. A task also contains a set of activities, $\mathcal{A}$, where each activity must be executed by one or more roles. The activities are defined as $\mathcal{A} = \{(a_1, \{r_1^1, r_2^1, ...\}), (a_2, \{r_1^2, r_2^2, ...\}), ...\}$, where $\forall r_i^j, r_i^j \in \mathcal{R}$, and a_i is the definition of the activity, e.g., including the title, description, goals, deliverables, etc., which is domain-specific.

Each role in the task is fulfilled by a compute unit. In some circumstances, e.g., for improving fault tolerance, multiple redundant compute units may also be assigned to a single role. Furthermore, for each role, we could also define functional and non-functional requirements.

The roles of the task, $\mathcal{R}$, is a set of roles containing their description and requirements, and $\mathcal{C}$ is a set of constraints representing non-functional requirements on the task-level, which both are discussed in the following subsection.

3.3.2 Task Requirement

We envisage an HCS, which allows consumers to specify their requirements that represent constraints for the compute units collectives provisioning and task execution. The task requirements defines the functional requirements, i.e., the capabilities required to perform a role in the task, as well the non-functional requirements.

Fuzzy Quality Requirement

In the context of human-based computing, due to the imprecise nature of human work, defining a precise constraint can be troublesome for consumers. Here, we propose to model quality requirements using *fuzzy concept* [146, 147]. For example, instead of saying "I need an electrical engineer with a problem solving qualification ≥ 0.75", the consumer could say "I need an electrical engineer with a *good* problem solving skill". For a given fuzzy quality q (e.g., $q = good$), we could measure the *fuzzy grade of membership* of a compute unit using the function $\mu_q : \mathbb{R}_{\geq 0} \rightarrow [0..1]$. This fuzzy quality concept can be applied to any functional and non-functional requirements when necessary depending

on the problem domain. Although fuzzy quality requirement can be applied to any non-discrete properties, in our work, we exemplify this fuzzy concept mainly to model the consumer requirements with respect to the capabilities and connectedness. The technique we use to utilize the fuzzy concept for provisioning compute units collectives is discussed in Section 5.3.1.

Functional Capability Requirement

A task contains one or more roles, which must be fulfilled to run activities defined to accomplish the task. For each role, the consumer defines a set of required capabilities. Our provisioning framework provisions a compute units collective for the task, where each member of the compute units collective with the required capabilities assigned to a role in the task.

More formally, a task t consists of a set of roles $\mathcal{R} = \{(r_1, \mathcal{C}p^{r_1}, \mathcal{C}^{r_1}), (r_2, \mathcal{C}p^{r_2}, \mathcal{C}^{r_2}), ...\}$. Here, r_i is the description of the role, e.g., title, presentation, etc., and $\mathcal{C}p^{r_i} = \{(cp_1, q_1), (cp_2, q_2), ...\}$ defines a set of required capabilities to perform role r_i, where cp_i defines the type of function that the role requires, and the required capability quality $q_i = 1$, for boolean capabilities, or $q_i = (0, 1]$, for non-discrete capabilities (see Section 3.2.2). Furthermore, a non-discrete capability quality can also be defined using the fuzzy qualities, e.g., $q_i = fair|good|very_good$.

$\mathcal{C}^{r_i}$ and $\mathcal{C}$ define the constraints representing non-functional requirements for role r_i and for task-level, respectively, as follows.

Non-Functional Requirement

Non-functional requirements define constraints, to which the compute units collective provisioning and the task execution must comply. Non-functional constraints can be applied on a role-level (i.e., $\mathcal{C}^{r_i}$ defines constraints for role r_i), which limits the selection of a compute unit for the role, and on a task-level (i.e., $\mathcal{C}$ defines constraints for the task), which controls the overall compute units collective provisioning and task execution. A set of constraints, $\mathcal{C}^{r_i}$ or $\mathcal{C}$, is defined as $\{(p_1, op_1, q_1), (p_2, op_2, q_2), ...\}$, where p_i is the required property type, op_i is a constraint compliant operator such as *less than*, *more than*, *equals to*, *between*, etc, and q_i is a constraining quality value(s), again, either $q_i = 1$, or $q_i = (0, 1]$, or $q_i = fair|good|very_good$.

Examples of non-functional requirements on a role-level include, but not limited to, the individual cost, the location, and also the availability of the compute unit. Another example of a non-functional requirement typically found for human-based compute units is the success rate, which can be measured from the history of previous task assignments.

A task-level requirement defines a constraint against an aggregated property of the compute units collective while executing the task. For example, we could define a cost limit constraint, which limits the sum of costs imposed by every participating compute units, i.e., given a task t with cost limit $costLimit$, the selected compute units collective members $\mathcal{U}' = \{u_1, u_2, ..., u_n\}$, must satisfy $\sum_{u \in \mathcal{U}} cost(u, t) \leq costLimit$.

Task:		
Description: Detecting facility breakdown, for object x, in location y		
Task-Level Non-Functional Contraints: - *deadline:* breakdown must be detected within 1 hour		
Activities:		
	Associated Roles	
- Human-based data collection	Data collector	
- Human-based data assessment	Data assesor	
- Hardware sensing	Hardware sensor, Sensor communication provider	
- Stream analytics	Stream analyzers	
Roles:		
	Functional Capabilities	*Non-Functional Contraints*
- Data collector	(human) all participating citizens	location = y
- Data assessor	(human) *good* assessment skill	location = y *good* acceptance rate
- Hardware sensor	(machine) sensor for object x	location = y
- Sensor communication provider	(machine) sensor network for object x	location = y
- Stream analyzers	(machine) software-based stream analytics service	response time $< z$ availability $>$ *99.9%*

Table 3.1: An example of a task specification

A *deadline* constraint is also a common non-functional requirement, which define the maximum time all participating compute units must complete their roles. The final completion time of a task also depends on the execution sequence of the activities (see Section 3.3.4). For example, if all the activities within the task are executed fully in parallel, given a task t with a deadline constraint *deadline*, the provisioned compute units collective, $\mathcal{U}' = \{u_1, u_2, ..., u_n\}$, must satisfy $\max_{u \in \mathcal{U}} time(u, t) \leq deadline$.

Furthermore, a *connectedness* constraint is also an example of task-level non-functional requirement, which defines how well the participating compute units should be interconnected to each others. We can also use a fuzzy linguistic variable to define a connectedness constraint, e.g., the consumer may say "I want to have a compute units collective with *fair* connectedness".

3.3.3 Task Sample

To illustrate our task model, Table 3.1 shows an example of a task specification with its roles, activities, and functional and non-functional requirements. We do not aim to propose a new formal visual or textual representation of a task. Instead, our task model serves as an abstraction to be used in building necessary building blocks for HCSs. For example, in our provisioning framework (Chapter 5) and reliability analysis framework (Chapter 7), we use mathematical models based on the task model presented in this section to develop our algorithms and frameworks. Furthermore, in our prototype implementation (Chapter 4), we use JSON notation to realize this task model.

3.3.4 Collective Dependency

The execution of tasks by compute units collectives may employ different patterns, which represent activity sequences, depending on the problem domain and on the runtime systems. Some examples of such patterns include (i) *single unit*, where a compute unit executes individually on different tasks, (ii) *pipeline*, where the members of a compute units collective execute the task sequentially one after another, (iii) *parallel*, where the task is split into subtasks, assigned to compute units collective's members, and the results are merged back after they finish (e.g., in [76]), (iv) *fault-tolerant*, where the task execution is made redundant and the result is selected from the aggregation of the results (e.g., in [148]), and (iv) *shared artifacts*, where the members of compute units collectives works collaboratively over some objects shared among all compute units collective's members (e.g., in [149]).

One of the effect of these patterns to the compute units collective provisioning is that each pattern may require different ways to measure the compute units collective's properties. For example, given a task t, and a compute units collective $\mathcal{U} = \{u_1, u_2, ..., u_n\}$, the response time of the pipeline pattern may be defined as $\sum_{u \in \mathcal{U}} time(u, t)$, while the response time of the parallel pattern may be defined as $\max_{u \in \mathcal{U}} time(u, t)$, where $time(u, t)$ is the time required by the compute unit u to execute the task t.

Here, we introduce a more robust model, *the collective dependency model*, to describe the execution of tasks by compute units collectives. We are motivated by the understanding that members of a compute units collective depend on each others in order to execute the task effectively. In particular, this model features the following traits:

i) it allows defining dependencies between activities,

ii) it defines the association of activities to the required roles,

iii) it defines which sources of compute units can be assigned to roles, and

iv) it allows the definition of alternative executions, i.e., alternative dependencies between activities, and alternative assignments of different compute units sources to roles.

When running a particular task, each constituent compute unit participates in a certain role by executing the assigned activity. Our model is based on the dependencies among compute units while performing activities to define the interrelationships between these compute units.

Each activity in a task provides a deliverable that can be consumed by other activities. Hence, each activity depends on all of its dependencies so that it can be successfully accomplished. Furthermore, we also introduce alternate activity dependencies, where an activity can be accomplished after at least n of its dependencies have provided the required deliverables.

In this collective dependency model, we define an activity dependency graph as an acyclic graph $\mathcal{G}_{dep} = (\mathcal{A}, \mathcal{E})$, where $\mathcal{A}$ is the set of activities executed by the compute units collective, and $\mathcal{E}$ is the set of dependencies between activities in $\mathcal{A}$. Furthermore,

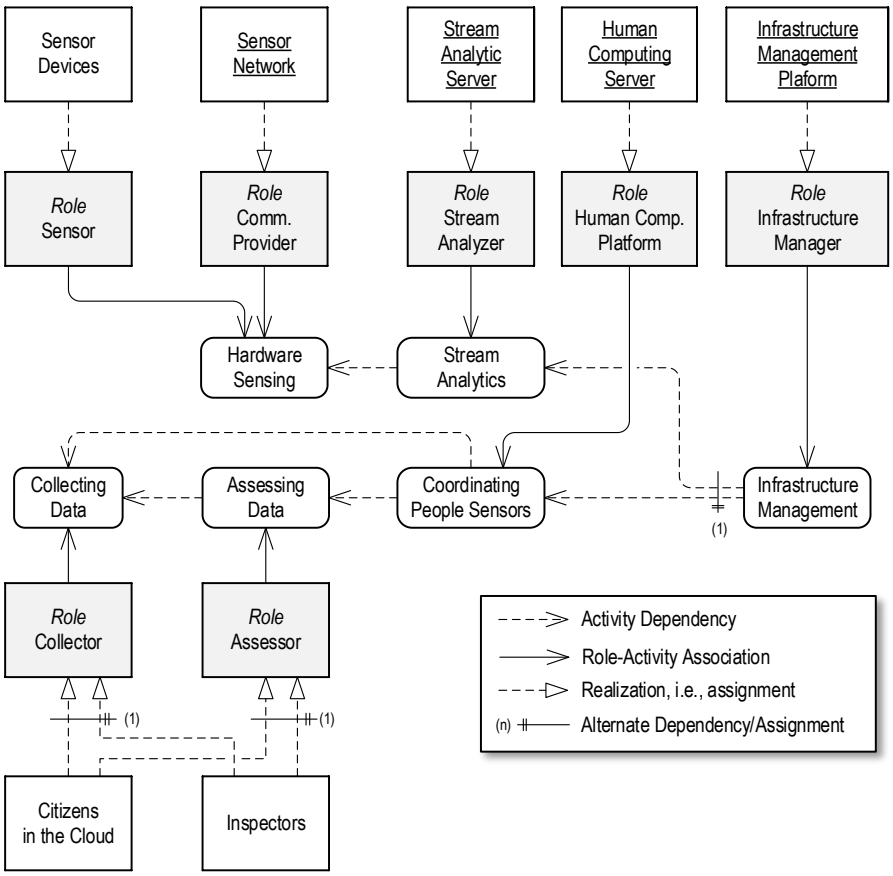


Figure 3.5: Collective Dependency

for each activity we define the role(s) associated to the activity, and for each role we define possible source(s) for the compute unit assignments, which can be a Cloud of compute units, or an on-premise compute units pool. Sources of compute units may represent either a fixed static set of compute units, or a group of compute units which are dynamically discovered and selected (see Section 3.2.1). Similar to alternate activity dependencies, alternate assignment sources can also be defined, where at least m assigned compute units from different sources must successfully perform the role.

The mechanism to obtain the collective dependency for a particular system is domain-specific. The application designer can define the collective dependency from the ground up, but it can also more practically be implied from the application design. For example, in a process-based application, such dependency can be inferred from the workflow. In a crowsdource-based application, the dependency can be deduced from the relationships between the microtasks, e.g., [76].

Returning to our previous scenario in infrastructure maintenance, in Fig. 3.5 we show an example of a collective dependency for detecting facility breakdown as well as the associated roles and possible sources of units assignments.

3.4 Chapter Summary

This chapter presents our models, which describe the characteristics of an HCS for executing tasks. First, we discussed architectural models of an HCS, which represent a coordination view and a runtime view of the system. Second, we defined the notion of compute units collectives representing groups of compute units assigned to execute tasks. Afterwards, we presented task models, which define the structure and the requirements of a task, as well as the dependency between activities within a task. These models lay out a foundation abstraction used in this thesis.

CHAPTER 4

Runtime and Analytics Platform for Hybrid Computing Systems

We have developed a platform, namely *Runtime and Analytics for Hybrid Computing Systems* (RAHYMS), as a prototype of our proposed architecture and models. This platform is open-source and available on GitHub[1].

The platform serves as a proof-of-concept to showcase a realization of a quality-aware and reliable HCS. Particularly, the platform is useful for (i) providing a tool for HCSs management, where the involved stakeholders could model, execute, and evaluate different system components and behaviors, (ii) providing a simulation testbed for evaluating HCS' building blocks.

This platform can be operated in two modes, i.e., (i) *interactive mode*, where compute units, tasks, and compute units collectives can be managed interactively, either programmatically trough provided APIs or manually using a web-based UI, and (ii) *simulation mode*, where compute units and tasks are generated and simulated based on used-defined configurations.

In the subsequent chapters, we use this platform to evaluate our proposed provisioning framework (Section 5.6), monitoring framework (Section 6.5), and reliability analysis framework (Section 7.4).

4.1 Prototype Architecture

We developed our platform based on the conceptual HCS runtime architecture discussed in Section 3.1.2. We realized the architecture by building runtime components using Java for both, interactive, and simulation modes. The main components of the architecture, discussed in Section 3.1.2, are reused for both interactive, and simulation modes. The prototype architecture for interactive mode is shown in Fig. 4.1, while the prototype

[1] `https://github.com/tuwiendsg/RAHYMS`

architecture for the simulation mode is depicted in Fig. 4.2. Here, the main components are shown as white boxes, while the components for interactive and simulation modes are shown as blue and green boxes, respectively.

Main Components

As a proof-of-concept, a task manager is implemented using a simple round-robin scheduler. The compute unit manager is developed to maintain the profiles of the compute units (i.e., their functional and non-functional properties and connectedness).

A runtime environment is created to manage the compute units collectives provisioned by the HCS. In a real-world implementation, such a runtime environment may also include features such as a collaboration platform, which enables exchanges of information among compute units in a running compute units collective.

The provisioning framework, the monitoring framework, and the reliability analysis framework are the main contribution of this thesis and discussed in Chapter 5, 6, and 7 respectively. The monitoring framework adopts an event-based approach (see Section 6.3) and is implemented using Esper[2] complex event processing (CEP) engine. And the adaptation engine utilized by the monitoring framework (see Section 6.4) is developed based on Drools rule engine[3].

Interactive Components

In the interactive mode, the task management, the unit management, as well as to the compute units collective management capabilities are exposed through REST APIs (refer to Appendix A.3.1 for more detail) developed using Jetty server[4] and Apache CXF framework[5]. A consumer application can interact with the platform through these APIs programmatically.

Additionally, we expose these capabilities on a web-based UI, developed using AngularJS framework[6], as a playground for the consumers. The sending of the tasks to compute units is achieved using SmartCom framework [150], which enables virtualized communication among diverse types of compute units.

Simulation Components

For the simulation mode, we developed a discrete-event simulation platform based on GridSim framework [151], which allows the execution of runtime components in a

[2] http://esper.codehaus.org
[3] http://www.drools.org/
[4] http://www.eclipse.org/jetty/
[5] https://cxf.apache.org/
[6] https://angularjs.org/

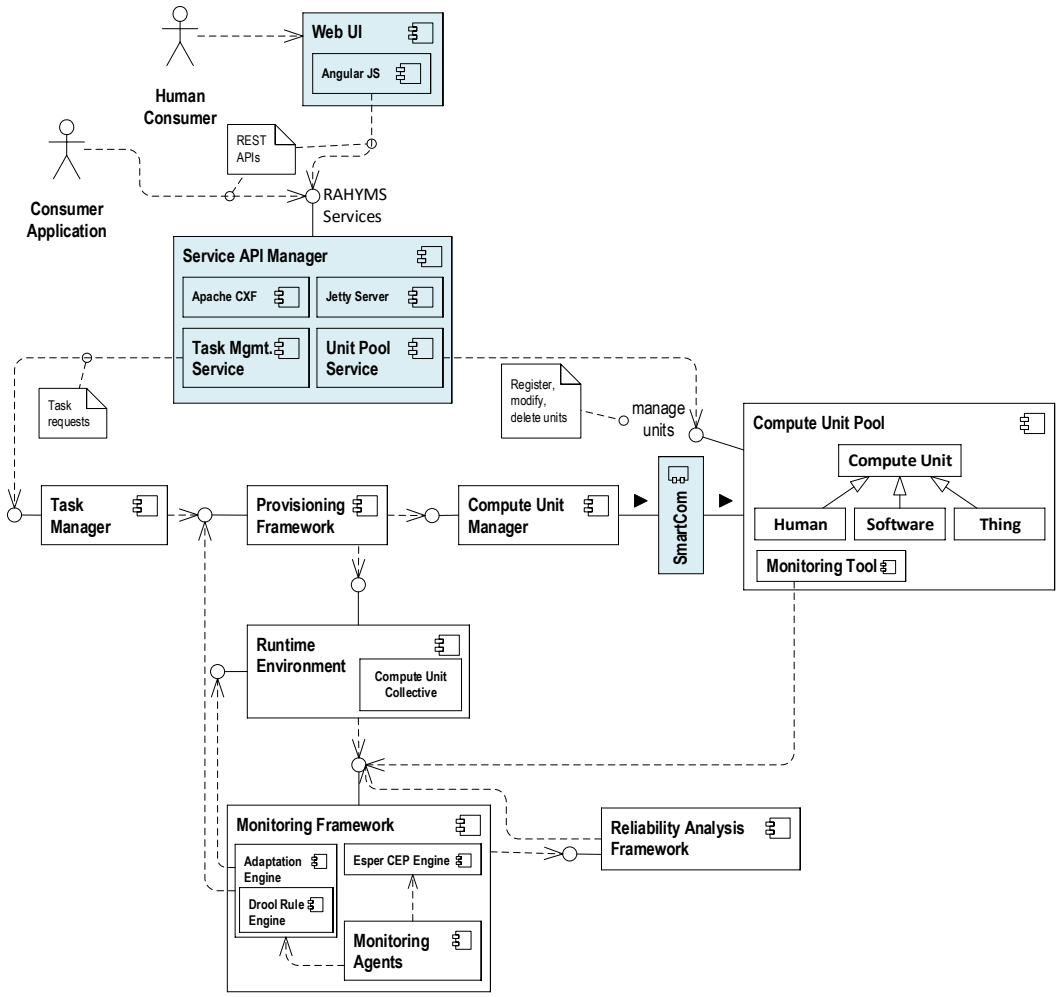


Figure 4.1: Prototype Architecture for Interactive Mode

simulated distributed environment. For this to work, we create GridSim adapters for each runtime components, i.e., the task manager, the provisioning framework, the compute unit manager, as well as all instances of compute units, so that they are running as parallel entities in a simulated grid-like setup.

Furthermore, our monitoring framework consists of monitoring agents, which retrieved events and metrics from other monitoring agents (refer to Section 6.3 for details). In the simulation mode, GridSim adapters are also attached to each monitoring agents.

The execution of the simulation is governed by configurations specified by the consumer (see Appendix A.2 for more detail). Prior to the simulation, the simulated cloud of compute units is populated with compute units generated based on the configuration. Afterwards, tasks are generated according to configuration. The compute units collectives are then provisioned to execute the tasks. The simulation terminates when all configured

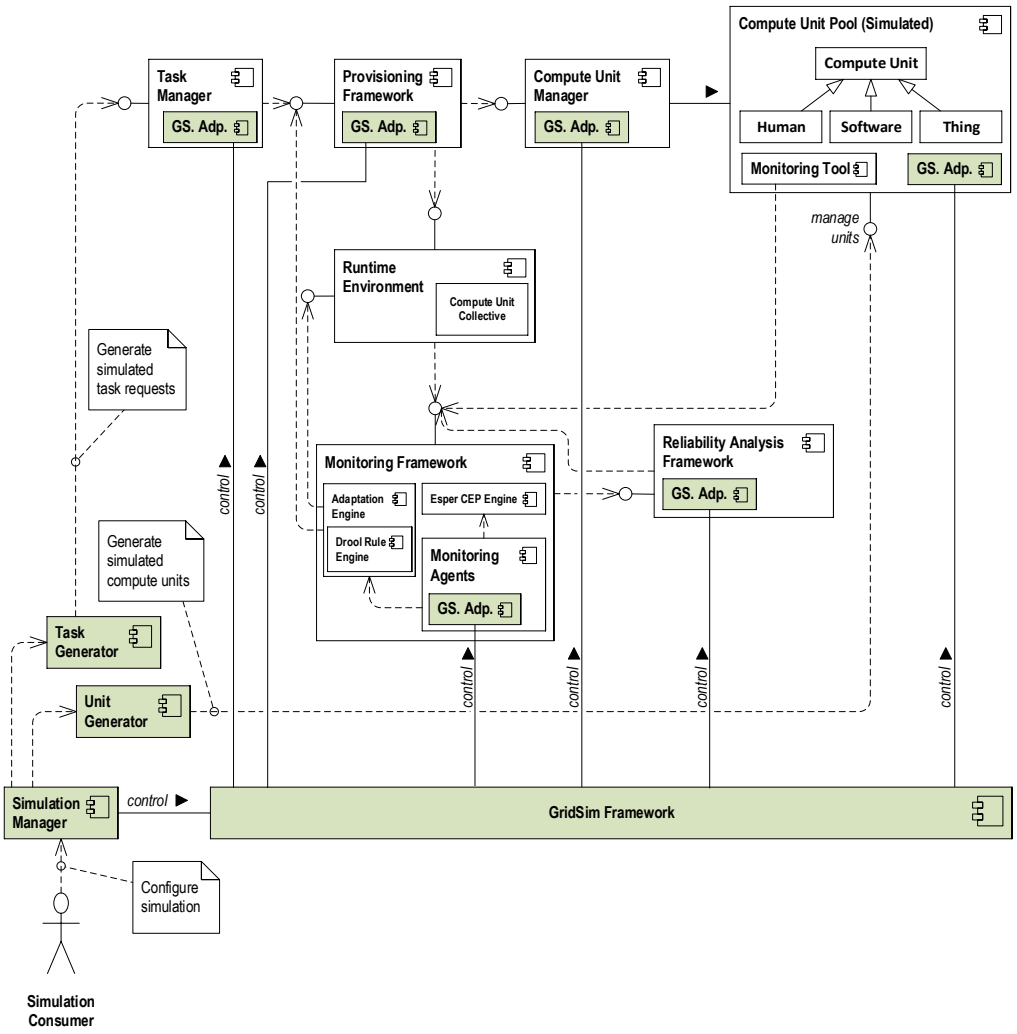


Figure 4.2: Prototype Architecture for Simulation Mode

tasks have been generated, and all the provisioned compute units collectives have finished executing the tasks.

4.2 Prototype Features

In this section, we highlight some key features of our platform, especially related to the main components, as well as the task modeling, unit management, execution trace, and simulation configuration. The simulation configuration feature is specific for the simulation mode, while others are for both interactive and simulation mode.

Task Modeling

Our platform allows consumers to model tasks with respect to the functional, and non-functional requirements, as well as the collective dependencies of the task as discussed in Section 3.3. The task model can be specified using APIs, for the interactive mode, or using the task generator configuration for the simulation mode.

Furthermore, to make task requests simpler for repetitive requests with similar model, we also allow task requests defined using only simple attributes, e.g., *tag* and *severity* attributes. Upon receiving such requests, our platform expands the request to a full-fledge task model, which already defined beforehand.

Unit Management

Our platform supports the management of both human-based and machine-based (software and things) compute units, which allow creating, retrieving, deleting, and modifying compute units with respect to their functional capabilities (i.e., services and skills provided by the compute unit), their non-functional properties, as well as their connectedness. These properties can be defined and stored internally in our platform, or retrieved from external platforms (e.g., from a crowdsoucing platform for human-based compute units) through available APIs. The unit management also features the capability to discovery compute units using filters, e.g., based on the functional capabilities of the compute unit.

Provisioning

Our platform provides various strategies for the provisioning of compute units collectives. Several formation algorithms are discussed in Section 5.4, e.g., greedy approach, and optimized formations using Ant Colony Optimization. Further formation strategies can be implemented and plugged into the platform. Details on these provisioning strategies are discussed in Section 5.3.

Monitoring and Adaptation

Our platform allows the creation of monitoring agents based on a JSON configuration. Some types of agents have been implemented according to the types of metrics they provide: agents that retrieve raw metrics via APIs, agents that process state-based events, and agents that process composite and correlated metrics using Esper Processing Language (EPL)[7].

Furthermore, we also implemented a type of monitoring agent capable of retrieving and replaying raw metrics from an available event log in CSV format. The implementation of monitoring agents also includes an adaptation engine, which can be utilized to perform adaptation actions when necessary (see Section 6.4). For each agent, a set of adaptation rules defined using Drools language can be provided.

[7] `http://www.espertech.com/esper/release-5.2.0/esper-reference/html/epl_clauses.html`

Reliability Analysis

Our platform provides a tool for performing reliability analysis on the running compute units collectives. The measurement of compute units collectives' reliability is performed based on the task model and the metrics of the running compute units. Reliability measurement is performed on each task execution, to allow studying how reliability metrics dynamically change over time. This is particularly useful in simulation mode, to understand how different task and provisioning models affect the reliability of the task execution. Details on the reliability analysis are presented in Chapter 7.

Execution Trace

Our platform is able to generate traces of the running system with respect to the task execution data, the compute units collective formation algorithm data, the monitoring data, and the reliability data. These traces are useful for further analysis, as well as generating insightful graphs. For example, the traces of the execution time of the compute units collective formation algorithms can be useful for comparing the performance of various algorithms.

Simulation Configuration

In the simulation mode, our platform can be configured using JSON configurations, which allow different scenarios to be simulated. These configurations can be specified to govern the generation of the simulated cloud of compute units with properties that are statistically distributed, e.g., to resemble a crowdsourcing marketplace or a social network for human-based compute units.

Furthermore, configurations for the tasks generator allow customization of the task requests, e.g., to define roles and collective dependencies. They can also be used to define statistically distributed functional and non-functional requirements of the tasks. During the formation of the compute units collectives for executing the task, these requirements will be matched with the properties of the generated compute units.

Details of these configurations is available in Appendix A.2. We include some pre-configured scenarios in the above-mentioned GitHub repository.

4.3 Chapter Summary

In this chapter, we presented a prototype implementation of a hybrid human-machine computing platform, namely *RAHYMS – Runtime and Analytics for Hybrid Computing Systems*.

This platform is designed based on the runtime model discussed in Section 3.1.2. The platform can be executed in two modes: interactive, and simulation mode. The interactive mode allows consumers to submit task requests to be executed by a provisioned compute units collective. The simulation mode, allow consumers to simulate a pool of compute units and a series of task requests with customizable configurations. This platform

provides a proof-of-concept as well as a simulation testbed for our main contributions of this thesis, i.e., the provisioning, monitoring framework, and reliability analysis framework, as discussed in the following sections.

CHAPTER 5

Provisioning

5.1 Introduction

Recently, we have been seeing on-demand online compute units provisioning models being applied not only to machine-based compute units, but also to human-based compute units. Unlike the machine-based compute units counterpart, quality control remains a major issue in provisioning human-based compute units. Current approaches for quality control are traditionally relies on simple and hard-wired techniques, which do not allow consumers to customize based on their specific requirements [36].

Our contribution presented in this chapter focuses on *Research Question 1: "How can we provide a collective of diverse compute units for executing tasks in a Hybrid Human-Machine Computing System (HCS) considering the consumer-defined requirements?"*. This contribution provides a flexible and quality-aware compute units collectives provisioning framework, which honors consumer-defined quality requirements, using compute units obtained either on-premise or from the Cloud, such as crowdsourcing marketplaces (for human-based compute units), Web service cloud (for software-based compute units), or IoT cloud (for thing-based compute units). In particular, our work presented in this Chapter provides a flexible provisioning framework, which allows using different formation techniques, and we present solution models which demonstrate the formation techniques for the framework. Specifically, we develop some algorithms, one of them based on the Ant Colony Optimization algorithm (ACO) approach, for dealing with the multiobjective quality-aware compute units collective formation problem. Moreover, our technique employs fuzzy concepts to deal with uncertain properties and requirements.

The proposed provisioning framework is particularly useful for, e.g., (i) providing a provisioning tool for the compute units collective management, which integrates the involved parties in the compute units collectives ecosystem, and (ii) providing a simulation testbed for studying various quality control technique for provisioning compute units collectives. To illustrate the usefulness of our framework, we study the feasibility of the

results using the ACO approach and compare with other simpler and common approaches, e.g., greedy and first-come-first-served strategies, using simulated experiments.

5.2 Provisioning Framework

Here, we extend the HCS runtime architecture presented in Section 3.1.2 and provide details of the provisioning framework. The core of our provisioning framework is the *provisioning middleware*, which connects the provisioning clients (e.g., the task manager, and the adaptation engine, see Section 4.1), the compute unit manager, the runtime environment, and the *formation engine*, as depicted in Figure 5.1. A scenario for a compute units collective provisioning starts when a provisioning client sends a provisioning request to the provisioning middleware. This request contains the consumer-defined functional and non-functional quality requirements for the task, such as the required capabilities of the compute units collective members, as well as their non-functional requirements, e.g., connectedness, maximum response time, and total cost.

The provisioning middleware discovers available and suitable compute units using a discovery service provided by the compute unit manager, which maintains the properties of the compute units from various compute unit clouds. This compute unit manager can also encapsulate different APIs provided by different compute unit clouds into a unified API. This compute unit manager also enables the provisioning of a compute units collective using different types of compute units from different clouds.

The formation engine is responsible for controlling the quality of the compute units collective formation. A *quality-aware provisioning strategy* is a strategy to control the formation of compute units collectives, which takes the consumer requirements and the properties of the discovered compute units into consideration. There are two types of quality-aware provisioning strategies covering two phases of the task' life cycle: pre-runtime and runtime. At pre-runtime, the quality-aware provisioning strategy governs the compute units collective formation. During runtime, a dynamic adaptation technique is employed to guarantee the required quality, which could trigger a re-provisioning request to replace one or more members of a running compute units collective.

A quality-aware provisioning strategy is implemented using an algorithm and executed by the formation engine. To process a task, the provisioning middleware requests the formation engine to form the compute units collective. Then, the formation engine invokes the algorithm to perform the formation. Upon receiving this formation, the provisioning middleware instantiate this compute units collective and deploy it to the runtime environment.

5.3 Quality-Aware Collective Formation Problem

Here, firstly we focus on pre-runtime quality-aware provisioning strategies, which deal with the formation of compute units collective prior to runtime. Later, we extend these strategies to support re-provisioning during task execution.

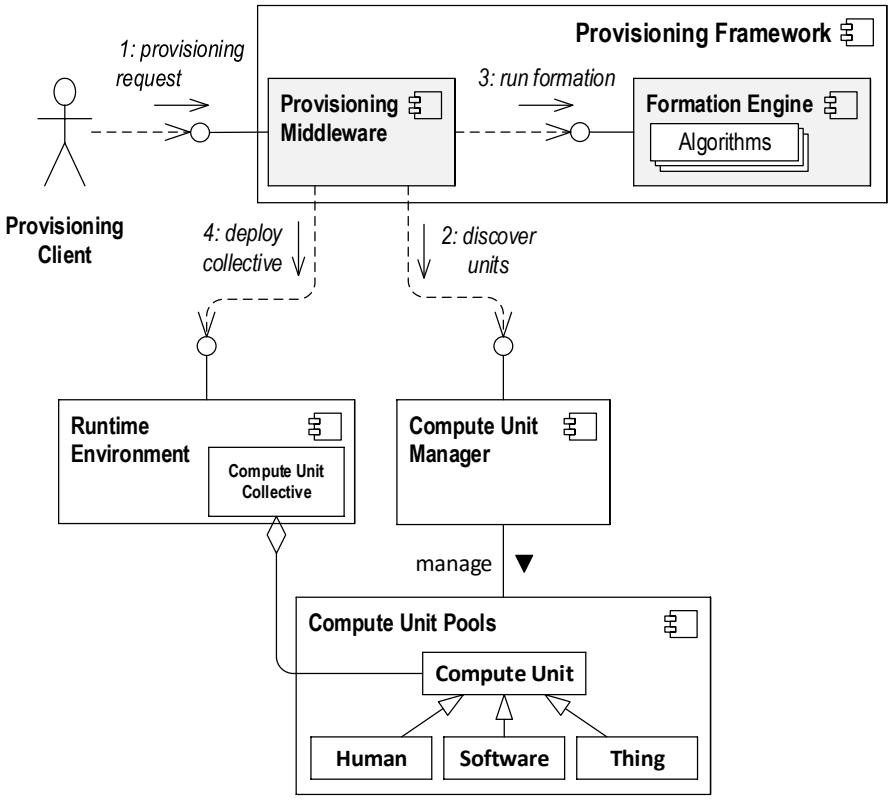


Figure 5.1: Compute Units Collective Provisioning Framework

We use the compute unit model and the task model as discussed in Section 3.2 and Section 3.3. Here, we summarize the notations of the models to be used throughout this chapter. A source of compute units contains a set of compute units, $\mathcal{U} = \{u_1, u_2, ..., u_n\}$, where $\forall u \in \mathcal{U}$ a set of functional capabilities, $\mathcal{C}p^u = \{(cp_1, x_1), (cp_2, x_2), ...\}$, can be defined. Each compute unit u also has a set of non-functional properties, $\mathcal{P}^u = \{(p_1, x_1), (p_2, x_2), ...\}$, where p_i is a property type and x_i is its value.

A task is a tuple, $t = (\mathcal{A}, \mathcal{R}, \mathcal{C})$, where $\mathcal{A}$ is a set of activities and their associated roles, $\mathcal{A} = \{(a_1, \{r_1^1, r_2^1, ...\}), (a_2, \{r_1^2, r_2^2, ...\}), ...\}$, $\forall r_i^j, r_i^j \in \mathcal{R}$, and $\mathcal{R}$ is a set of roles, $\mathcal{R} = \{(r_1, \mathcal{C}p^{r_1}, \mathcal{C}^{r_1}), (r_2, \mathcal{C}p^{r_2}, \mathcal{C}^{r_2}), ...\}$, where r_i is the description of the role, and $\mathcal{C}p^{r_i}$ is functional capability of the role, i.e., $\mathcal{C}p^{r_i} = \{(cp_1, q_1), (cp_2, q_2), ...\}$, where cp_i represents a capability type, and q_i represents the required capability quality (i.e., $q_i = 1$, or $q_i = (0, 1]$, or $q_i = fair|good|very_good$ for fuzzy requirements). The task-level constraints $\mathcal{C}$ and role-level constraints $\mathcal{C}^{r_i}$ is defined as a set of tuples $\{(p_1, op_1, q_1), (p_2, op_2, q_2), ...\}$, where p_i is a property type, op_i is a constraining (e.g., comparison) operator, and q_i is a constraining value(s).

Furthermore, for the purpose of provisioning optimization, the task t may also contain a consumer-defined provisioning objective, $\mathcal{O}$, which represents the weighting factors that a quality-aware provisioning strategy must take into account to optimize the properties of

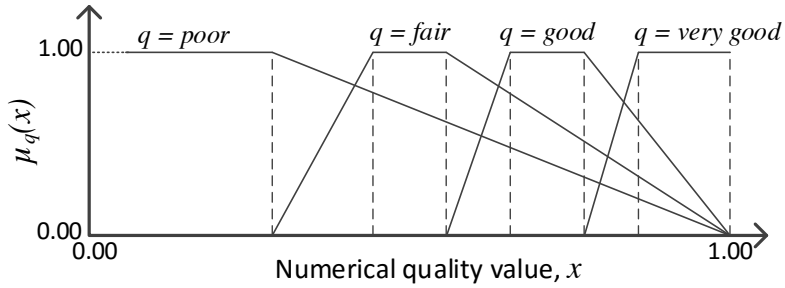


Figure 5.2: An Example of Fuzzy Grade of Membership Functions

the formed compute units collectives. Without loss of generality, we focus on the weighting factors $\mathcal{O} = (w_{cp}, w_{cn}, w_{time}, w_{cost})$, where $w_{cp}, w_{cn}, w_{time}, w_{cost}$ are the weighting factors for the functional capabilities, connectedness, response time, and cost, respectively, as discussed in Section 3.2.2.

We formulate the compute units collective formation problem, which takes the quality requirements from the consumer into consideration, and propose some algorithms to solve it. Given a set of compute units, $\mathcal{U}$, which are connected in a graph $G = (\mathcal{U}, \mathcal{E})$, and a task request $t = (\mathcal{A}, \mathcal{R}, \mathcal{C}, \mathcal{O})$, we define the *compute units collective formation problem* as a problem of finding $\mathcal{U}' \subset \mathcal{U}$ as members of a compute units collective for executing task t which optimizes the properties of $\mathcal{U}'$ according to weighting factors $\mathcal{O}$ for fulfilling all roles $r_i \in \mathcal{R}$, subjects to task-level constraints $\mathcal{C}$ and all role-level constraints $\mathcal{C}^{r_i}$. In the following we discuss some building blocks required to solve the compute units collective formation problem.

5.3.1 Measuring Fuzzy Qualities

As discussed in Section 3.3.2, we employ fuzzy concept [146] to model capabilities for executing jobs and connectedness of the formed compute units collectives. Given a numerical quality value x of a functional or non-functional property of a compute unit (e.g., capability level, connectedness, etc), we could measure the *grade of membership* of the value for a given fuzzy quality q (e.g., *poor, fair, good*) using the function $\mu_q(x) : \mathbb{R}_{\geq 0} \rightarrow [0..1]$. An example of grade of membership function is the *trapezoidal* membership function [112], where the grade of membership function for each fuzzy quality resembles a trapezoid shape. Fig. 5.2 shows an example of a set of grade of membership functions adopted from the trapezoidal membership function.

A task contains a set of roles $\mathcal{R}$, where for each $r_i \in \mathcal{R}$, we have a set of required capabilities $\mathcal{C}p^{r_i} = \{(cp_1, q_1), (cp_2, q_2), ...\}$. In the case where fuzzy quality values are used for q_i, the formation engine attempts to find a set of compute units $\mathcal{U}' = \{u_1, u_2, ...\}$, which maximizes $M_{r_i}(u_i)\ \forall r_i \in \mathcal{R}$. M_{r_i} represents the aggregated grade of membership on the *intersection* of the fuzzy sets of all required fuzzy qualities in the role, i.e., given $\mathcal{C}p^{r_i} = \{(cp_1, q_1), (cp_2, q_2), ...\}$, we define

$$M_{r_i}(u) = \wedge_{(cp_k,q_k)\in\mathcal{C}p^{r_i}}\{\mu_{q_k}(x_k^u)\}, \tag{5.1}$$

where μ_{q_k} is a grade of membership function for quality q_k, and x_k^u is the numerical capability level of compute unit u for capability type cp_k. Here, we use the `min` operation as the interpretation of fuzzy set intersection [147]. For non-fuzzy capability requirements, the formation engine could simply try to maximize $\sum x_k^u$.

Similarly, for the connectedness requirement, given a connectedness quality q_{conn} (e.g., *poor*, *fair*, or *good* connectedness), the formation engine composes a compute units collective $\mathcal{U}' = \{u_1, u_2, ...\}$ with a connectedness graph $G' = (\mathcal{U}', \mathcal{E}')$, which maximizes $\mu_{q_{conn}}(conn(G'))$, where $conn(G')$ is given by Equation 3.1.

5.3.2 Construction Graph

We approach the above defined compute units collective formation problem as a graph path finding problem. Given a set of compute units $\mathcal{U} = \{u_1, u_2, ...\}$ and a set of required roles $\mathcal{R} = \{r_1, r_2, ...\}$, the solution space is formulated using a *construction graph*, i.e., an acyclic directed graph, where each node represent a solution component as shown in Fig. 5.3. A solution component, $sc_{i,k}$ is a tuple (r_i, u_k), which represent an assignment of role r_i to compute unit u_k, where $r_i \in \mathcal{R}$ and $u_k \in \mathcal{U}$. Directed edges are created from $sc_{i,k}$ to $sc_{j,k}$, for all $u_k \in \mathcal{U}$, where $\forall(i,j), 1 \leq i < j \leq |R|$. Additionally, we add two sentinel nodes, sc_0 and sc_F, as start- and end-nodes respectively, where sc_0 has outgoing edges to all $sc_{1,k}$ and sc_F has incoming edges from all $sc_{|R|,k}$, for all $u_k \in \mathcal{U}$. Hence, the compute units collective formation problem is a path finding problem from sc_0 to sc_F.

A solution of the problem $\mathcal{S}$, i.e., a path from sc_0 to sc_F, represents a set of assignments for each role in $\mathcal{R}$. For example, a path $sc_0 \rightarrow sc_{1,a} \rightarrow sc_{2,b} \rightarrow ... \rightarrow sc_{m,n} \rightarrow sc_F$ represents a set of assignments $\mathcal{S} = \{(r_1, u_a), (r_2, u_b), ..., (r_m, u_n)\}$.

In our provisioning framework, the compute unit manager maintains a set of connected compute units $G = (\mathcal{U}, \mathcal{E})$ obtained from various compute unit sources. The goal of an algorithm for solving the formation problem is to find an optimized solution $\mathcal{S}$ in the search space $\mathcal{R} \times \mathcal{U}$. Due to the size of $\mathcal{U}$ obtained from compute unit clouds, this search space can be extremely huge. Therefore, we filter out non-feasible solution components based on the feasibility of each compute unit for each role. The filtering can be done using the discovery service provided by the compute units manager based on the required capabilities $\mathcal{C}p^{r_i}$ and non-functional constraints $\mathcal{C}^{r_i}$ for each role r_i.

However, this filtering does not guarantee a full feasibility of complete assignments on all jobs. To guide our heuristic algorithms for selecting assignments towards a feasible solution while optimizing the objective, we define two algorithm control mechanisms: *the local fitness* which represents the fitness of an assignment relative to other possible assignments for the same role, and *the objective value of a solution* which represents the fitness of a complete solution. The formulation of these mechanisms is stimulated by the necessity to measure the heuristic factors and solution quality in ACO approaches[152]. However, as we show in Section 5.4, these mechanisms can also be used by other heuristics, e.g., greedy approaches.

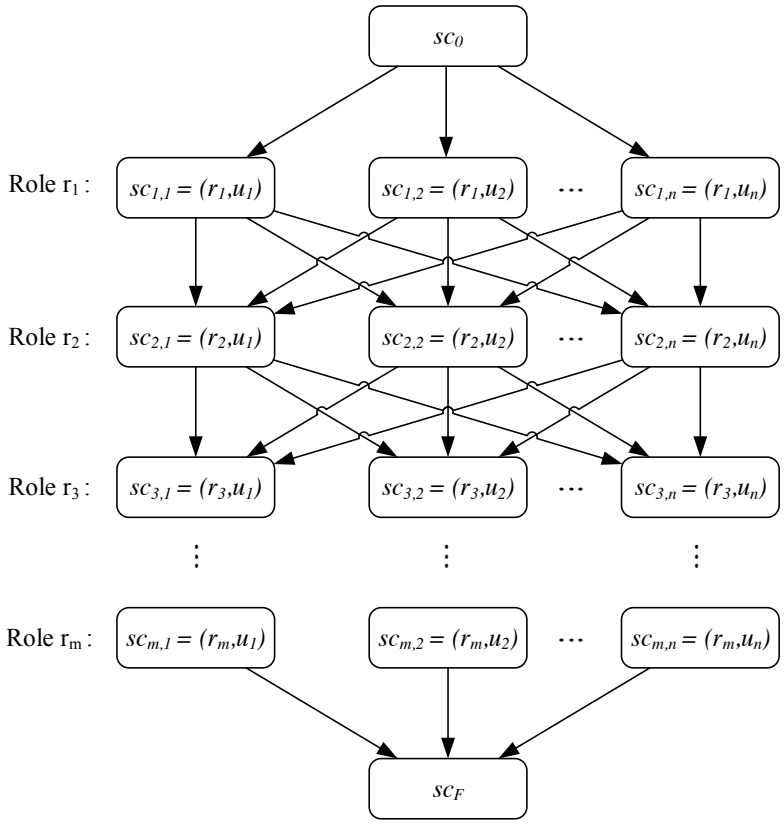


Figure 5.3: Construction Graph for Collective Formation Problem

5.3.3 Local Fitness

The local fitness of an assignment is defined based on partially selected assignment, starting form an empty set of assignments when the algorithm begins. Given a task t with the objective weighting factors $\mathcal{O} = (w_{cp}, w_{cn}, w_{time}, w_{cost})$, a set of selected partial assignments up to role number $i-1$, $\mathcal{S}_{i-1}$, that already contains a set of compute units $\mathcal{U}_{i-1}$, and a set of possible assignments for the subsequent role r_i, $\mathcal{S}_i^P$ (i.e., after a filtering is performed), the *local fitness* λ for an assignment (i.e., a solution component) $sc_{i,j} = (r_i, u_j)$, $sc_{i,j} \in \mathcal{S}_i^P$, is defined as

$$\lambda(sc_{i,j} \cup \mathcal{S}_{i-1}) = \frac{\lambda_{cp} \cdot w_{cp} + \lambda_{cn} \cdot w_{cn} + \lambda_{time} \cdot w_{time} + \lambda_{cost} \cdot w_{cost}}{w_{cp} + w_{cn} + w_{time} + w_{cost}} \tag{5.2}$$

where λ_{cp}, λ_{cn}, λ_{time}, and λ_{cost} respectively represent the local fitness with respect to the capability, connectedness, time, and cost, which are given by

$$\lambda_{cp}(sc_{i,j} \cup \mathcal{S}_{i-1}) = M_{r_i}(u_j),$$
$$\lambda_{cn}(sc_{i,j} \cup \mathcal{S}_{i-1}) = \frac{conn(u_j \cup \mathcal{U}_{i-1}) - conn(\mathcal{U}_{i-1})}{\gamma_{conn} + conn(u_j \cup \mathcal{U}_{i-1}) - conn(\mathcal{U}_{i-1})},$$
$$\lambda_{time}(sc_{i,j} \cup \mathcal{S}_{i-1}) = \frac{\gamma_{time}}{\gamma_{time} + time(u_j \cup \mathcal{U}_{i-1}, t) - time(\mathcal{U}_{i-1}, t)},$$
$$\lambda_{cost}(sc_{i,j} \cup \mathcal{S}_{i-1}) = \frac{\gamma_{cost}}{\gamma_{cost} + cost(u_j, t)},$$

where M_{r_i} is the aggregated grade of membership function for role r_i defined in Eq. 5.1, *conn* is a connectedness function such as defined in Eq. 3.1, *time* is a response time estimation function such as discussed in Section 3.2.2, and γ_* is an adjustable parameter, e.g., we can use the consumer-defined *costLimit* as γ_{cost}, and *deadline* as γ_{time}. Note that these local fitness values are normalized, i.e., $\lambda : \mathcal{S}^P \mapsto [0..1]$.

5.3.4 Objective Value of Solution

For each solution $\mathcal{A}$, i.e., a complete set of assignments for all roles in $\mathcal{R}$, we could measure the normalized *objective value* returned by the function $f : \mathcal{S}^D \mapsto [0..1]$, $\mathcal{S}^D = \mathcal{R} \times \mathcal{U}$. Given an objective weighting factors $\mathcal{O} = (w_{cp}, w_{cn}, w_{time}, w_{cost})$, the objective function $f(\mathcal{S})$ for $\mathcal{S} = \{(r_1, u_1), (r_2, u_2), ...\}$, is defined as follows:

$$f(\mathcal{S}) = 1 - \frac{f_{cp}(\mathcal{S}) \cdot w_{cp} + f_{cn}(\mathcal{S}) \cdot w_{cn} + f_{time}(\mathcal{S}) \cdot w_{time} + f_{cost}(\mathcal{S}) \cdot w_{cost}}{w_{cp} + w_{cn} + w_{time} + w_{cost}}, \tag{5.3}$$

where

$$f_{cp}(\mathcal{S}) = \wedge_{(r_i,u_i)\in\mathcal{S}}\{M_{r_i}(u_i)\},$$
$$f_{cn}(\mathcal{S}) = \mu_{q_{conn}}(\mathcal{U}_\mathcal{S}),$$
$$f_{time}(\mathcal{S}) = \frac{\gamma_{time}}{\gamma_{time} + time(\mathcal{U}_\mathcal{S}, t)}, \text{ and}$$
$$f_{cost}(\mathcal{S}) = \frac{\gamma_{cost}}{\gamma_{cost} + \sum_{u_i \in \mathcal{U}_\mathcal{S}} cost(u_i, t)}.$$

$\mathcal{U}_\mathcal{S}$ is the set of compute units in assignments $\mathcal{S}$, i.e., for any $\mathcal{S} = \{(r_1, u_1), (r_2, r_2), ..., (r_n, u_n)\}$, $\mathcal{U}_\mathcal{S} = \{u_1, u_2, ..., u_n\}$. For $f_{cp}(\mathcal{S})$, we again apply `min` function as the interpretation of intersection operation $\wedge$. The fuzzy grade of membership function for connectedness, $\mu_{q_{conn}}$, has been discussed in Section 5.3.1, and γ_* constants uses the same values as in local fitness calculation. The function $time(\mathcal{U}_\mathcal{S}, t)$ returns the aggregated response time of all compute units in $\mathcal{U}_\mathcal{S}$ discussed in Section 3.2.2. The goal of a compute units collective formation algorithm is to minimize $f(\mathcal{S})$.

5.4 Formation Algorithms

We have established the building blocks required for solving the compute units collective formation problem. Here, we present some algorithms to solve the compute units collective

formation problem.

Simple Algorithms We present two simple algorithms that can be used to find a solution of the compute units collective formation problem based on the *First Come First Selected algorithm (FCFS)* and the *greedy* approach.

FCFS Approach This approach resembles the approach traditionally used in task-based crowdsourcing model: the first compute unit who 'bids' wins the task. Assuming that a standby compute unit is interested in taking a task, we select the first earliest available compute unit for each job. In the case where there are some compute units with the same earliest availability, we pick one randomly. Such approach is also typically used in a round-robin scheduling strategy of machine-based compute units.

Greedy Approach Initially we construct a solution by selecting assignments for each job that has the highest local fitness value. Afterwards, we gradually improve the solution by changing an assignment at a time. Improvement is done by randomly selecting a job, and randomly selecting another compute unit for that job. If the new assignment improve the objective value of the solution, we replace the associated old assignment with this new better one. This procedure is repeated until a certain number of maximum cycle is reached. The greedy approach makes a locally optimized choice for each job at a time with a hope to approximate the global optimal solution.

Ant Colony Optimization Ant Colony Optimization algorithm (ACO) is a meta-heuristic inspired by the foraging behavior of some ant species[152]. In the ACO technique, artificial ants tour from one node to another node in the solution space until a certain goal is achieved. The tour is guided by the pheromone trails, which are deposited by the ants to mark the favorable path. The nodes visited in a complete tour represent a solution. Once all ants have finished a tour, the process is repeated for a specified number of cycles or until a certain condition is met. The best solution of all cycles is selected as the solution of the problem.

An ant starts a tour on the construction graph from sc_0, then travels to the next nodes (r_1, u_i), ... (r_n, u_n) until reaches sc_F, hence all roles $r_i \in \mathcal{R}$ are assigned. Each node has a probability to be selected determined by the pheromone trails and the heuristic factor of the node.

Several variants of ACO algorithms have been proposed. Here, we develop our algorithm based on three variants: the original Ant System algorithm (AS) [153], $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System algorithm (MMAS) [154], and Ant Colony System algorithm (ACS) [155]. Generally, the ACO approach is depicted in Algorithm 5.1.

When traveling through the nodes, at each move i, an ant k constructs a partial solution $\mathcal{S}_i^k$ consisting all visited nodes for roles 1 to i. When ant k has moved $i-1$

times, the probability it moves to another node (r_i, u_j) is given by

$$p_{i,j}^k = \begin{cases} \dfrac{(\tau_{i,j})^\alpha \cdot (\eta_{i,j})^\beta}{\sum_{(r_i,u_w)\in\mathcal{S}_i^{P'}} \left((\tau_{i,w})^\alpha \cdot (\eta_{i,w})^\beta\right)} & \text{if } (r_i, u_j) \in \mathcal{S}_i^{P'}, \\ 0 & \text{otherwise,} \end{cases} \tag{5.4}$$

where $\mathcal{S}_i^{P'} = \mathcal{S}_i^P - \mathcal{S}_{i-1}^k$, i.e. the set of possible assignments for role r_i containing only compute units that are not yet included in $\mathcal{S}_{i-1}^k$; $\tau_{i,j}$ is the pheromone value of the node (r_i, u_j) at the current cycle; and $\eta_{i,j} = \lambda(sc_{i,j} \cup \mathcal{S}_{i-1}^k)$ is the heuristic factor as defined in Equation 5.2. The relative importance of pheromone and heuristic factor are determined by parameter α and β. ACS variant uses a modified transition rule, i.e., *pseudorandom proportional rule* as shown in [155].

At the end of each cycles, pheromone trails on all nodes are updated. At each cycle t, given the number of ants $nAnts$, the basic pheromone update formula for a node (r_i, u_j), which is proposed by the original AS variant [153], is given by

$$\tau_{i,j}(t) = (1-\rho) \cdot \tau_{i,j}(t-1) + \sum_{k=1}^{nAnts} \Delta\tau_{i,j}^k, \tag{5.5}$$

where $\rho \in (0..1]$ is the *pheromone evaporation* coefficient, and $\Delta\tau_{i,j}^k$ is the quantity of pheromone laid by ant k on the node (r_i, u_j), which is given by

$$\Delta\tau_{i,j}^k = \begin{cases} {}^{Q}/_{f(\mathcal{S}^k)} & \text{if } (r_i, u_j) \in \mathcal{S}^k \wedge \mathcal{S}^k \text{ is feasible,} \\ 0 & \text{otherwise,} \end{cases} \tag{5.6}$$

where $\mathcal{S}^k$ is the solution found by ant k and Q is an adjustable parameter. $\mathcal{S}^k$ is feasible if it does not violate any task-level constraints $\mathcal{C}$ (see Section 3.2). Note that up to this point, we have already filtered out component solutions for each role r_i that violate role-level constraints $\mathcal{C}^{r_i}$ (see Section 5.3.2). Here, we exclude solutions that violate one or more task-level constraints so that only feasible solutions are promoted by the ants. The pheromone update for MMAS and ACS variant has the same principle but different formula as presented in [154] and [155].

5.5 Runtime Re-Provisioning

So far, we have discussed to quality-aware compute units collectives formation problem prior to runtime. During runtime, one or more currently assigned compute units may need to be replaced, e.g., due to an adaptation request 6.4. In this case, a re-provisioning request is sent to provisioning middleware. The provisioning strategies discussed above needs to be adjusted to handle a re-provisioning request.

Given a set of roles $\mathcal{R}$, a re-provisioning request is a request for fulfilling a subset of roles $\mathcal{R}' \subset \mathcal{R}$. We approach such re-provisioning request by modeling the solution space using a *pruned* construction graph. For each unchanged roles r_i, i.e., $r_i \in \mathcal{R} \wedge r_i \notin \mathcal{R}'$, we

Algorithm 5.1: Ant-based Solver Algorithm

```
initialize graph and pheromone trails
repeat
    A_ants ← ∅
    for i = 0 to nAnts do
        A ← find a tour for ant_i
        A_ants ← A_ants ∪ A
    end
    update pheromone trails
until ∃A ∈ A_ants f(A) = 0 or is stagnant or max cycles reached
```

remove all the possible solution components, except the one with previously provisioned compute units.

For example, let us take a look again at the construction graph for finding a set of optimized assignments as shown in Fig. 5.3. Consider we have provisioned the compute units collective with $\mathcal{S} = \{(r_1, u_\alpha), (r_2, u_\beta), (r_3, u_\gamma), ..., (r_l, u_\lambda), (r_m, u_\mu)\}$. During runtime, a re-provisioning request for role $\mathcal{R}' = r_2, r_m$ arrives. For handling the request, we create again the construction graph as before, but this time for all $r_i \notin r_2, r_m$, we only add the previously provisioned compute units, i.e., $\{(r_1, u_\alpha), (r_3, u_\gamma), ..., (r_l, u_\lambda)\}$. Such a pruned construction graph is shown in Fig. 5.4.

Once we have the (pruned) construction graph for the re-provisioning request, we employ again the same constructs and algorithms described previously for the regular construction graph.

5.6 Evaluation

In this section, we present our experiments to evaluate our provisioning technique using our prototype platform presented in Chapter 4. Our platform allows different provisioning strategies to be implemented into the provisioning framework. Here, we implemented various quality-aware provisioning strategies discussed in Section 5.3, i.e., first-come-first-selected (FCFS) approach, greedy approach, and ACO-based approaches. Here, our goal is to study the dynamic of the provisioning system with respect to various provisioning strategies. For this purpose, we generate a simulated pool of compute units, and send simulated task requests to the provisioning middleware.

In our experiments, we focus on the following aspects of the compute units collective provisioning: (i) we study our quality-aware provisioning strategies based on the three aforementioned algorithms and analyze the performance and result, and (ii) we study the ACO approach to have an insight of (a) the effect of different algorithm parameters (b) the performance and result of the three different ACO variants.

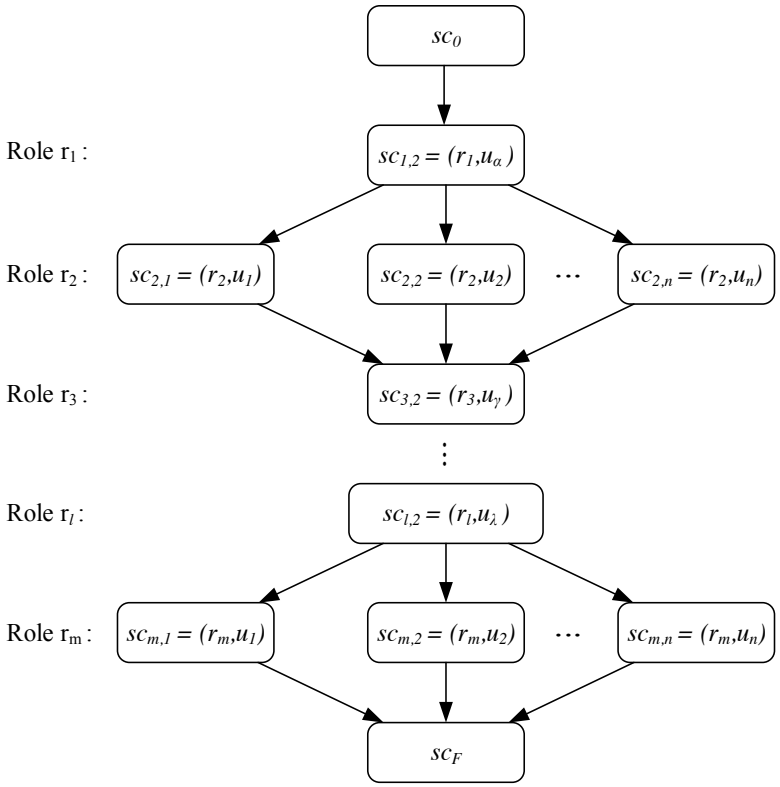


Figure 5.4: Pruned Construction Graph for Re-provisioning

Experiments Setup

Our prototype compute unit manager maintains a work queue for each compute unit. Each compute unit can only execute a single job at a particular time. We experiment with *parallel* pattern (see Section 3.2), where subtasks, i.e., jobs, are assigned to the compute units collective members and executed in parallel. We generate 500 compute units on our simulated pool. We define 10 types of functional capabilities, and each compute unit is randomly endowed with these capabilities. The consumer application generates task request with random parameters. Each job in a task has a set of functional capabilities requirements with the required fuzzy quality uniformly distributed over four fuzzy quality levels: *poor*, *fair*, *good*, and *very good*. In this experiment, we use the fuzzy grade of membership functions adopted from [112] as shown in Fig. 5.2, which support *over-qualification* when assigning compute units collective members. Over-qualification in a fuzzy quality evaluation allows selecting a compute unit with a higher fuzzy quality, e.g., selecting a *good* compute unit for a *fair* quality requirement.

Algo	Average Objective Values ($\overline{f}$)	Average Capability Levels ($\overline{f_{cp}}$)	Average Response Times	Violation	Algorithm Time
FCFS	0.4501	0.0810	6.06	4%	0.9117 ms
Greedy	0.3468	0.2130	11.87	0%	0.1219 s
AS	0.3147	0.3228	10.90	0%	6.6565 s

Table 5.1: Formation algorithms' results and performance comparison

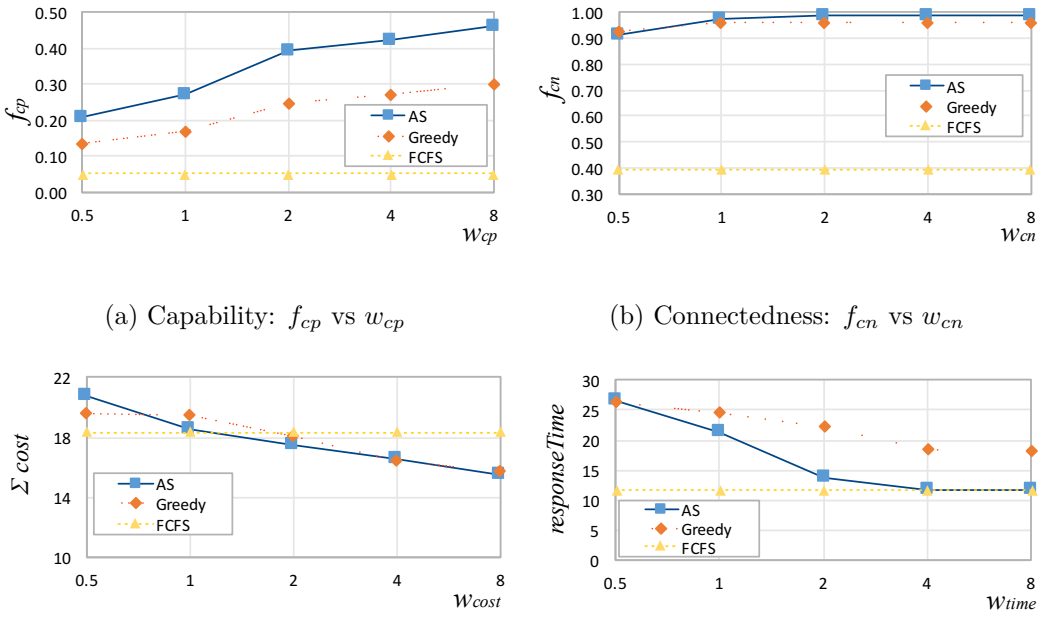
Experiment 1 - Comparing Quality-aware Provisioning Strategies

To study our quality-aware provisioning strategies, we configure our consumer application to randomly generate and submit 100 task requests. We repeat the same setup three times to test the formation engine configured using the three implemented algorithms: the FCFS algorithm, the greedy algorithm, and the original variant of Ant System (AS) algorithm.

Table 5.1 shows a comparison of average results from all task requests. The AS algorithm outperforms the others with respect to the aggregated objective, i.e., minimizing $f(\mathcal{A})$. The AS algorithm also provides compute units collective formation with better capability levels. However, as expected, the FCFS algorithm gives the fastest running time. But considering the nature of human tasks, few seconds running times of the AS algorithm and the greedy algorithm are reasonable. This fast performance is not without cost, since the FCFS algorithm concludes a solution too fast considering the response time only, it results in some constraint violations. Fortunately, due the filtering of the search space (see Section 5.3.2), violations on capability constraints do not occur.

Experiment 2 - Objective-based Quality Control

We are also interested in studying the quality control behavior with respect to the objective weightings, $\mathcal{O} = (w_{cp}, w_{cn}, w_{time}, w_{cost})$, as defined by the consumer. Figure 5.5 shows results of our experiment using task requests with varying objective weightings and compute units collective size. On each experiment shown on the sub-figures, we vary one weight from 0.5 to 8 and fix the others. The results show that the AS algorithm honors the consumer defined weights better compared to the other two. The sensitivity of the FCFS algorithm is flat on all cases, because it does not consider the objective weightings during the formation. The sensitivity levels of the cost weight w_{cost} of the greedy algorithm and the AS algorithm are similar, due to the fact that the local fitness value for cost λ_{cost} contributes linearly to the objective value of the cost f_{cost}. For the connectedness sensitivity, the AS algorithm cannot be seen clearly outperforms the greedy algorithm, because the formed compute units collective almost reach the upper limit of f_{cn}, i.e., 1.



(a) Capability: f_{cp} vs w_{cp}

(b) Connectedness: f_{cn} vs w_{cn}

(c) $\sum_{v_i \in \mathcal{V}_\mathcal{A}} cost(v_i, t)$ vs w_{cost}

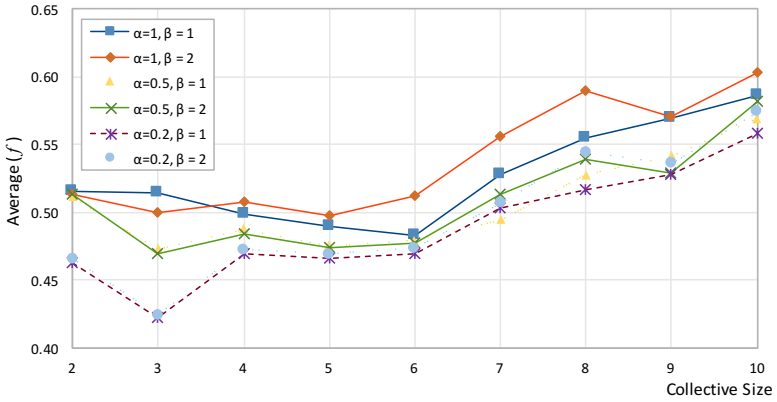
(d) $responseTime(\mathcal{V}_\mathcal{A}, t)$ vs w_{time}

Figure 5.5: Sensitivity on objective weightings

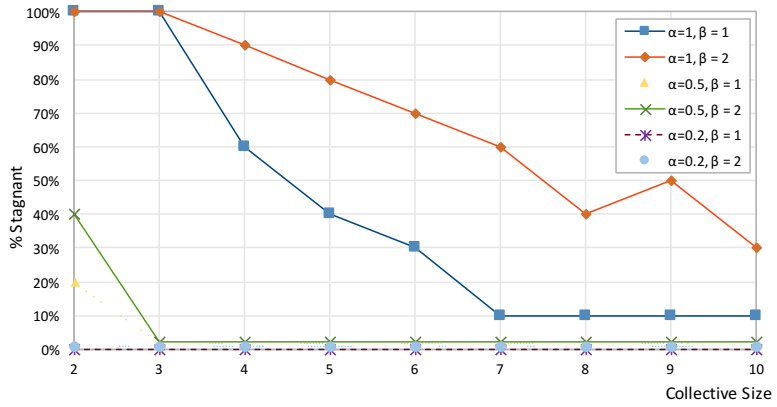
Experiment 3 - Comparing ACO Approaches

Knowing that the AS algorithm provides better results in many aspects, we carry out further experiments to understand the behavior of our ACO approach. First, we study the effect of the ACO parameters to the performance and to the quality of the resulted compute units collective formation. In our experiment, we use the AS variant and fix the pheromone evaporation factor low, $\rho = 0.01$. If ρ is set too high, it will cause the pheromone trails to be negligible too fast. Then, we vary the relative importance of pheromone and heuristic factor, α and β. Figure 5.6a shows how different α and β yield different results with respect to the average aggregated objective value of the best compute units collectives formed. Furthermore, we run the experiments for 8 ants in 2000 cycles and see whether a stagnant behavior occurs as shown in Figure 5.6b. A cycle is said to be stagnant when all ants result in the same compute units collective formation; hence, causing the exploration of the search space to stop. Our experiments show that the combination of $\alpha = 0.2$ and $\beta = 1$ gives best results.

Furthermore, we extend the experiment further using the same α and β parameters to the other two ACO variants. We are interested in finding out which ACO variants give faster conclusion to a good compute units collective formation. We run the experiment using 8 ants and 10000 cycles as shown in Figure 5.7. The result shows that the MMAS variant gives better compute units collective formations (less objective values) in less number of cycles than the others.



(a) Objective values average



(b) Stagnant behavior

Figure 5.6: Influence of α and β

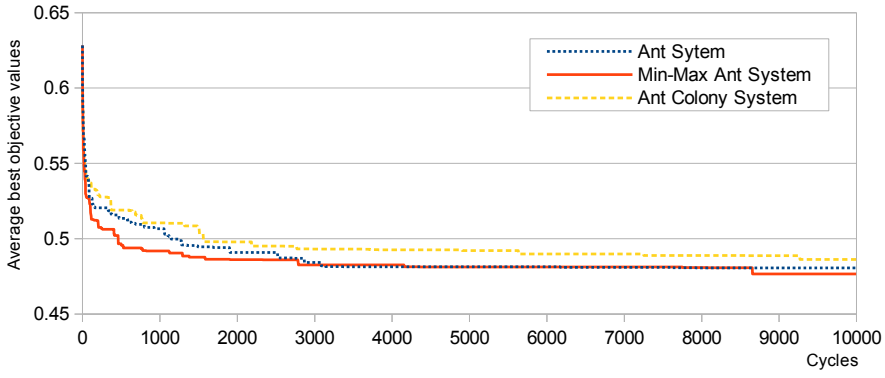


Figure 5.7: Comparison on results of ACO variants

5.7 Chapter Summary

In this chapter, we presented our approach for the provisioning of compute units collectives according to the functional and non-functional quality requirements from the consumers. We proposed some algorithms for finding optimized formations of compute units collectives considering both, consumer-defined quality requirements, and properties of the discovered compute units.

Different quality-aware provisioning strategies implemented by different algorithms cater different needs. In our experiments setup, we showed an ACO based algorithm provides better results in some aspects. However, there is no "one size fits all" strategy. For example, the FCFS approach may be preferable in some circumstances where the response time is the most important factor and the consumer only cares about capability constraints, which happens in typical microtask crowdsourcing systems. The usefulness of our framework is therefore also to support multiple strategies.

CHAPTER 6

Monitoring

6.1 Introduction

Monitoring and analyzing metrics in Hybrid Human-Machine Computing Systems (HCSs) are inevitable steps to improve the quality of task executions. Monitoring tools provide insights to plan, manage, and adapt the systems to fulfill quality requirements.

However, the diversity of involved compute units in HCSs introduces challenges for monitoring such systems. Existing monitoring systems traditionally deal with homogeneous compute units, for example, infrastructure/platform monitoring systems (e.g., [118, 119]), software-based services monitoring systems (e.g., [39, 156]), and IoT monitoring systems (e.g., [157]). Moreover, different types of compute units have different lifecycles, which require different measurement techniques, monitoring cycles and events. Different types of compute units may also have similar metrics, but different semantics interpretation, which needs to be correlated. Furthermore, monitoring HCSs require us to interface the underlying diverse resources as well as the application platforms. To the best of our knowledge, currently no monitoring system exists that deals with thing-based, software-based, and human-based compute units in an integrated manner.

Our contribution presented in this chapter addresses *Research Question 2: "How can an HCS with diverse metrics models and diverse subsystems be effectively monitored?"*. Our goal is to provide a generic monitoring framework for HCSs, which captures and processes metrics from diverse compute units, e.g., sensors, actuators, gateways, software services, and human-based compute units. In this contribution, particularly (i) we propose metric models to handle metrics with different semantics that are necessary for characterizing behaviors of compute units in various HCSs, (ii) we bring into effect the notion of Quality of Data (QoD) for monitoring data enabling effective monitoring in HCSs, and (iii) we propose a framework and implement a prototype of a monitoring system for HCSs.

6.2 Metrics and Quality of Data

A monitoring system centers around metrics, which need to be captured, analyzed, and delivered to the clients. Existing metric constructs in monitoring systems need to be extended in order to engage with the dynamics of HCSs. Furthermore, the concept of Quality of Data (QoD) can be leveraged to deal with the problem of different metrics qualities, as well as different monitoring requirements in HCSs. In this section, we discuss various metric suitable for HCSs, how to measure them, and the application of QoDs in HCS monitoring.

6.2.1 Metrics

Useful Metrics for HCSs

Metrics of an HCS may contain aggregation of metrics from machine-based compute units (i.e., from the thing-based, and software-based systems) and human-based compute units. Metrics traditionally found for machine-based compute units may have the equivalent for human-based compute units with similar meaning. In Table 6.1 we show some metrics and some possible definitions of the metrics for machine-based and human-based compute units as well as the aggregation definition for HCSs, where u is a compute unit and t is a task. Other definitions may also be employed according to the problem domain.

Metric Measurement

To capture the dynamics of an HCS, we define four classes of metrics, namely *raw metrics*, *composite metrics*, *state metrics*, and *correlation metrics*. The first two classes of metrics are commonly found in monitoring system, e.g., [127, 156]. However, due to the diversity of HCSs, we introduce the state metrics and correlation metrics.

Raw metrics are metrics that capture information from the underlying resources. These metrics can be collected using different means depending on the underlying monitors. For example, typical raw metrics from a cloud service can be obtained using an exposed API. Generally, raw metrics can be obtained in two manners, by pulling periodically, or by using a publish-subscribe approach. Similarly, to the typical machine-based computing platforms, in human-based computing platforms, some raw metrics may be directly provided by the platform (e.g., acceptance rates and locations of human-based compute units in a crowdsourcing marketplace).

Composite metrics can be defined using an arithmetic expression, an aggregate function, or a custom composite function of other metrics. For example, the utilization of a thing-based system containing a set of sensors can be measured by aggregating the number of sensors actively sending streams of data in a particular time frame, e.g., using moving average aggregation on a sliding window.

Metrics	Machine-based Definitions	Human-based Definitions	HCS Aggregation	Description
$Util(u)$	$CPUUsage(u)$	$\frac{Active(u)}{\text{MAXACTIVE}}$	$\frac{\sum_{\forall u \in \mathcal{U}} Util(u)}{\lvert\mathcal{U}\rvert}$	$Util(u)$ = utilization of unit u, $Active(u)$ = the total duration of active time in the past 24 hours of unit u, MAXACTIVE = threshold for maximum active time per human unit, $\mathcal{U}$ = the set of units in the HCS
$RT(t,u)$	$FT(t,u) - AT(t,u)$	$FT(t,u) - AT(t,u)$	$\max_{\forall u \in \mathcal{U}} FT(t,u) - \min_{\forall u \in \mathcal{U}} AT(t,u)$	$RT(t,u)$ = response time for task t by unit u, FT = finish time, AT = assignment time
$Cost(t,u)$	$CostT(u) \cdot RT(t,u)$	$CostA(t,u)$	$\sum_{\forall u \in \mathcal{U}} Cost(t,u)$	$Cost(t,u)$ = cost for executing task t by unit u, $CostT$ = cost per time unit, $CostA$ = cost per task assignment

Table 6.1: Metric Examples

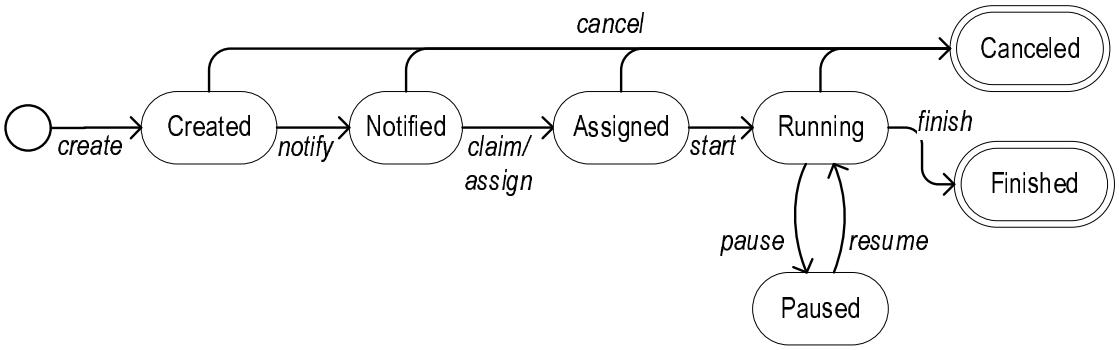


Figure 6.1: An Example of State Transitions for Human-based Tasks

State metrics define measurements related to the state transitions during runtime. In typical human-task and process-based systems, we often deal with the underlying monitoring tools that are capable of capturing events representing transitions from one runtime state to another, e.g., [122]. We use state metrics for capturing metrics related to the state transitions, such as how many times a compute unit enters a particular state, how long a compute unit stays in a state for a particular time window, and so on. Consider, for example, a human-based task running on a process-based system [29]. On a particular process instance, the human-based task may be transient from one state to another, such as shown in Fig. 6.1. Using a finite-state automata model, we can then define some primitives for a given entity e (e.g., a compute unit or a task) and a state s. For example, we could define primitives $time(e, s)$, $count(e, s)$, $duration(e, s)$, which present the last timestamp e enters s, the number of times e enters s, and the total duration of e staying in s respectively. We can also extend these primitives to perform the measurement on a particular time window, e.g., the last 24 hours. Furthermore, a composite metric can then be defined using these primitives, e.g., $time(t, Finished) - time(t, Created)$ defines the response time of task t.

Correlation metrics allow computing together metrics with different semantics from different sources. We support this type of metrics to tackle the problem of combining together metrics from diverse compute units. To correlate diverse metrics we need to specify three things: the sources of metrics or events that we want to correlate, the normalization function that should be applied so that the source metrics have uniform semantics before we combine them together, and the aggregation function to calculate the value of the new correlated metric.

Complex Metric Samples

To demonstrate these metric classes, consider how we can measure the *utilization* metrics of the system in infrastructure maintenance scenario discussed in Section 1.2. For example, on the thing-based and software-based systems, the utilization of sensor i, $SensorUtil(i)$,

and the utilization of machine j running a data analytic service, $MachineUtil(j)$, can be obtained using raw metrics, typically using a certain API exposed by the platform.

However, for human-based systems, the utilization measurement can be more complicated. First of all, we have to define what *utilization* means for human-based compute units. In the case of humans, there is no notion of CPU usage as traditionally found for software-based systems. Without loss of generality, let us, for example, define the utilization of a human-based compute unit as the time the human-based compute unit spent for executing all assigned tasks in a given time window w. Hence, using the state metrics, we can the measure the utilization of human-based compute unit k as $HumanUtil(k) = duration(k, Running, w)$.

Often we need to monitor system-wide correlated metrics instead of metrics for a particular human-based compute unit. For example, it can be necessary to see the overall average utilization from all the three subsystems (i.e., the human-based, software-based, and thing-based systems). Or we can monitor the *top-k* units with the highest utilization, regardless they are thing-based, software-based, or human-based compute units, so that to identify bottlenecks. To obtain such metrics, we need to combine together $SensorUtil$, $MachineUtil$, and $HumanUtil$ metrics, and resolve any semantics differences among them. This is where our correlation metric model becomes practical. Firstly, we could normalize the *HumanUtil* metric so that it has the same value range and it has an acceptable similar meaning compared to the $SensorUtil$ and $MachineUtil$ metrics. One reasonable normalization of the human utilization against the machine utilization is to set a maximum threshold of working time that a human-based compute unit may work in the past 24 hours, that is $HumanUtil'(k) = duration(k, Running, 24hours)/\text{MAX}$. This definition surely is not the sole definition of human utilization, different definitions may be applied according to the problem domain.

6.2.2 Quality of Data

Collecting and processing monitoring data on cloud-based large scale systems introduces an inherent problem, that is, a huge number of monitoring data lead to high network utilization and heavy data processing. On the contrary, the human-based computing counterpart is typically running in a much slower pace due to longer life-cycles, e.g., assignments to a single human-based compute unit may take place in the order of minutes, hours, or even days.

We apply the concept of *Quality of Data* (QoD) [126] allowing monitoring clients to specify the monitoring requirements as a trade-off for resources usages or costs. Such QoD-aware monitoring solves the above-mentioned problems in two ways: (i) it allows the monitoring clients or providers to request or produce monitoring data on a lower quality level to reduce costs, and (ii) it allows interweaving monitoring data on different subsystems having different QoD into similar QoD, hence it becomes reasonable to correlate metrics from those subsystems. We discuss the interpretation of QoD in the context of HCS monitoring and some use-cases of such QoD as follows.

QoD Interpretation for HCS Monitoring

We focus on three QoD measures, namely *accuracy*, *freshness*, and *data rate* (or *rate* for short) as defined in the following paragraphs and illustrated in Fig.6.2.

Data Rate The data rate of a monitoring data, $Rate(d)$, represents the frequency on which the monitoring agent should report the data. Many techniques can be used to obtain data on any particular time point, e.g., to use last actual retained data or to use moving average values. When the real data has a lower data rate, the monitoring agent may perform techniques, e.g., an interpolation technique, for estimating the data in-between.

Accuracy The accuracy of monitoring data is derived from the difference between the true value of the data with the value last reported to the client, i.e., given a data, d, the accuracy of the data is defined as $Acr(d) = |v(d') - v(d)|$, where $v(d)$ is the actual value of d and $v(d')$ is its last reported value.

Freshness The freshness of monitoring data defines the timing skew between the true timestamp of the data and the timestamp when the data is reported, i.e., it is defined as $Frs(d) = t(d') - t(d)$, where $t(d)$ is the actual timestamp of d and $t(d')$ is the report timestamp.

More formally, given a QoD requirement, $\mathcal{Q} = (R_R, R_A, R_F)$, where R_R is a data rate requirement, R_A is an accuracy requirement, and R_F is a freshness requirement, the monitoring tool must deliver a set of reported data $\mathcal{I}$ from the actual set of data $\mathcal{J}$, that fulfills the following constraints:

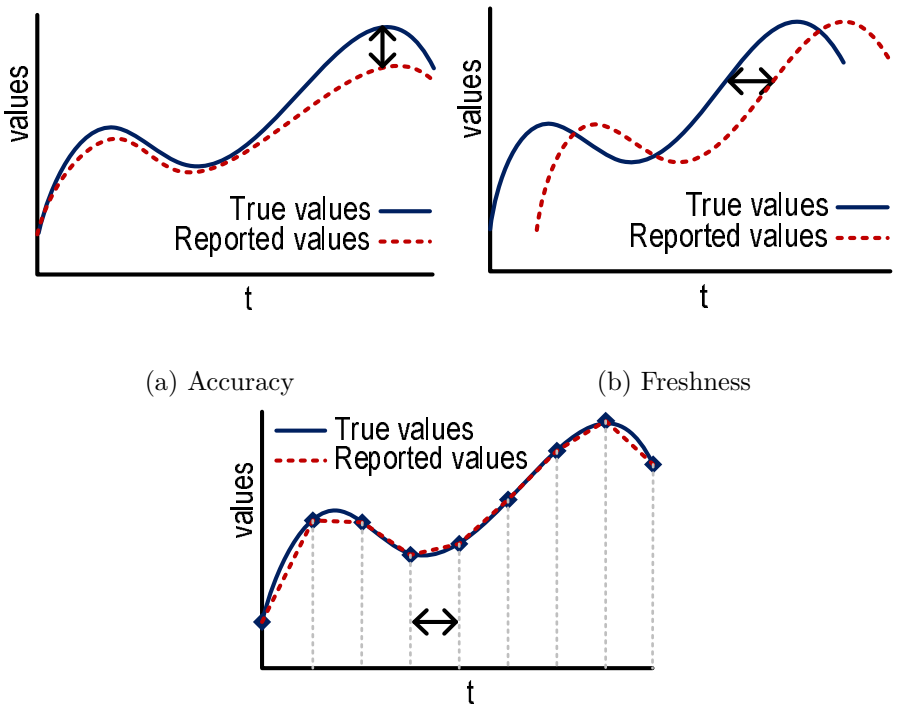
$$\begin{aligned} \forall d' \in \mathcal{I}, \forall d \in \mathcal{J}, \quad & t(d'_{i-1}) < t(d_j) \leq t(d'_i) \implies \\ & \Big(t(d'_i) - t(d'_{i-1}) \leq R_R \quad \wedge \\ & |v(d'_i) - v(d_j)| \leq R_A \quad \wedge \\ & t(d'_i) - t(d_j) \leq R_F \Big), \end{aligned} \tag{6.1}$$

where $t(d)$ is the time when the data d is sent, and $v(d)$ is the value of the data d.

QoD-aware Monitoring Usages

The usages of QoD-aware monitoring can be seen from two perspectives. First, from the perspective of a monitoring provider, QoD-aware monitoring helps to increase efficiency on resource usage, e.g., data bandwidth. Second, it allows a monitoring client to define more precisely the quality of data they need.

Consider, for example, a human client who wants to monitor system utilization, but she/he does not want the monitoring reports to be intrusive. Hence, she/he may want to request utilization data for one hour intervals. However, she/he does not want to miss



(a) Accuracy (b) Freshness

(c) Data Rate

Figure 6.2: Quality of Data in HCS Monitoring

rapid changes on the system utilization. Hence, she/he puts in a data quality requirement that the accuracy of the data she/he receives should not be more than 0.10 points. In this case, the monitoring system delivers the data on (maximum) an hourly rate, but also makes sure that the last reported data does not differ more than 0.10 points from the real value.

QoD-Aware Data Delivery

Based on QoD requirements, a monitoring provider may provide a QoD-aware data delivery by optimizing monitoring resources while still fulfilling the constraints as described in Eq. 6.1. There are many ways to achieve such QoD-aware data delivery, e.g., depending on the optimization objective. We present an example of such QoD-aware data delivery algorithm (see Algorithm 6.1), which minimizes the number of messages (i.e., the number of sent monitoring data), while still honoring the QoD requirements. This algorithm defers the sending of data, retain it, and calculate the right time to send the data according to the data rate, R_R, accuracy, R_A, and freshness, R_F, requirements. Here, the RECEIVE function is executed when a monitoring consumer (see Section 6.3) receives data, and the SEND function sends data to the subscriber.

Algorithm 6.1: Algoritm for QoD-Aware Data Delivery

Input: QoD requirements, $\mathcal{Q} = (R_R, R_A, R_F)$

```
1  Procedure receive(data) /* invoked when consumer receives data */
2      retain(data, MaxRetained)
3      if R_A is set then
4          if |data − lastSentData| > R_A then
5              send(data)
6              SCHEDULER.cancelPreviousWorker()
7              return
8          end
9      end
11
12     if R_F is set then
13         if not dataChanged ∧ data ≠ lastSentData then
14             dataChanged ← True
15             nextWakeTime ← NOW + R_F
16             SCHEDULER.cancelPreviousWorker()
17             SCHEDULER.wakeMeAt(nextWakeTime)
18         end
19     end
21
22     if R_R is set then
23         if nextWakeTime > NOW + R_R then
24             nextWakeTime ← NOW + R_R
25             SCHEDULER.cancelPreviousWorker()
26             SCHEDULER.wakeMeAt(nextWakeTime)
27         end
28     end
29 end
31
32 Procedure wake()
33     data ← estimateFromRetainedData() send(data)
34     if rate is set then
35         nextWakeTime ← NOW + rate
36         SCHEDULER.wakeMeAt(nextWakeTime)
37     end
38 end
```

6.3 Distributed Monitoring Framework

Our monitoring framework consists mainly of *monitoring agents* (or agents for short), which provide events and metrics for other monitoring agents, as shown in Fig. 6.3 (here, a metric is a type of event, in the remainder of this chapter we use them interchangeably). Such a distributed and recursive nature of the monitoring agents structure allows our framework to scale according to the scale of the HCS.

Our framework adopts an event-based approach using the publish/subscribe pattern. Each agent publishes topics that contain metric values for other agents. Each agent can either subscribe to certain topics from other agents, or retrieve metrics from their own adapters connecting to the underlying monitoring tools. Eventually, a client application (or a client, for short) can then consume metrics from one or more agents and use it in the application logic. Fig. 6.3 also represents an example of agents topology.

In the following subsections we discuss the construct of monitoring agents and the communication protocol between those monitoring agents, as well as some technical considerations for agents' implementation.

6.3.1 Monitoring Agent

A monitoring agent is a software component containing a *monitoring producer* (MP), which produces events and metrics according to the context it monitors. Inputs of a monitoring agent come from one or more *monitoring adapters* (MA), which retrieve events and metrics from the underlying resources or application monitors, and/or one or more *monitoring consumers* (MC), which consume events from other agents.

The monitoring adapter (MA) component of an agent, adapts events and metrics captured from a specific monitoring tool provided by the application or resource platform. An MA may retrieve metrics through an underlying protocol provided by the monitoring tool. For example, the presence events of a human-based service can be provided using XMPP. Other publish/subscribe protocols, such as AMQP and MQTT may also be used for retrieving metrics from software-based or thing-based systems. Other underlying platforms may also utilize other techniques to propagate metrics such as using polling techniques, e.g., [120], or multicast techniques, e.g., [119]. Hence, the implementation of MA is platform-specific, and is beyond the scope of our work. Moreover, instead of implementing an MA for an underlying monitoring tool, an agent may also consume metrics provided by another agent by implementing a monitoring consumer (MC).

Note that the proposed construct of monitoring agents is a conceptual abstraction. On the practical level, multiple agents can be implemented either on a single physical node (e.g., an agent may consume its own metrics to produce more complex metrics), or on multiple nodes. In the case where agents are distributed, they need to communicate each others. The communication protocol between MCs and MPs is discussed in Section 6.3.2.

A proposed implementation model of an agent is shown in the bottom-right inset of Fig. 6.3. Here, an agent is implemented using a complex event processor to process event streams retrieved via an MC. A straightforward raw metric can publish directly from the incoming event stream. For composite and correlation metrics, an associated event query

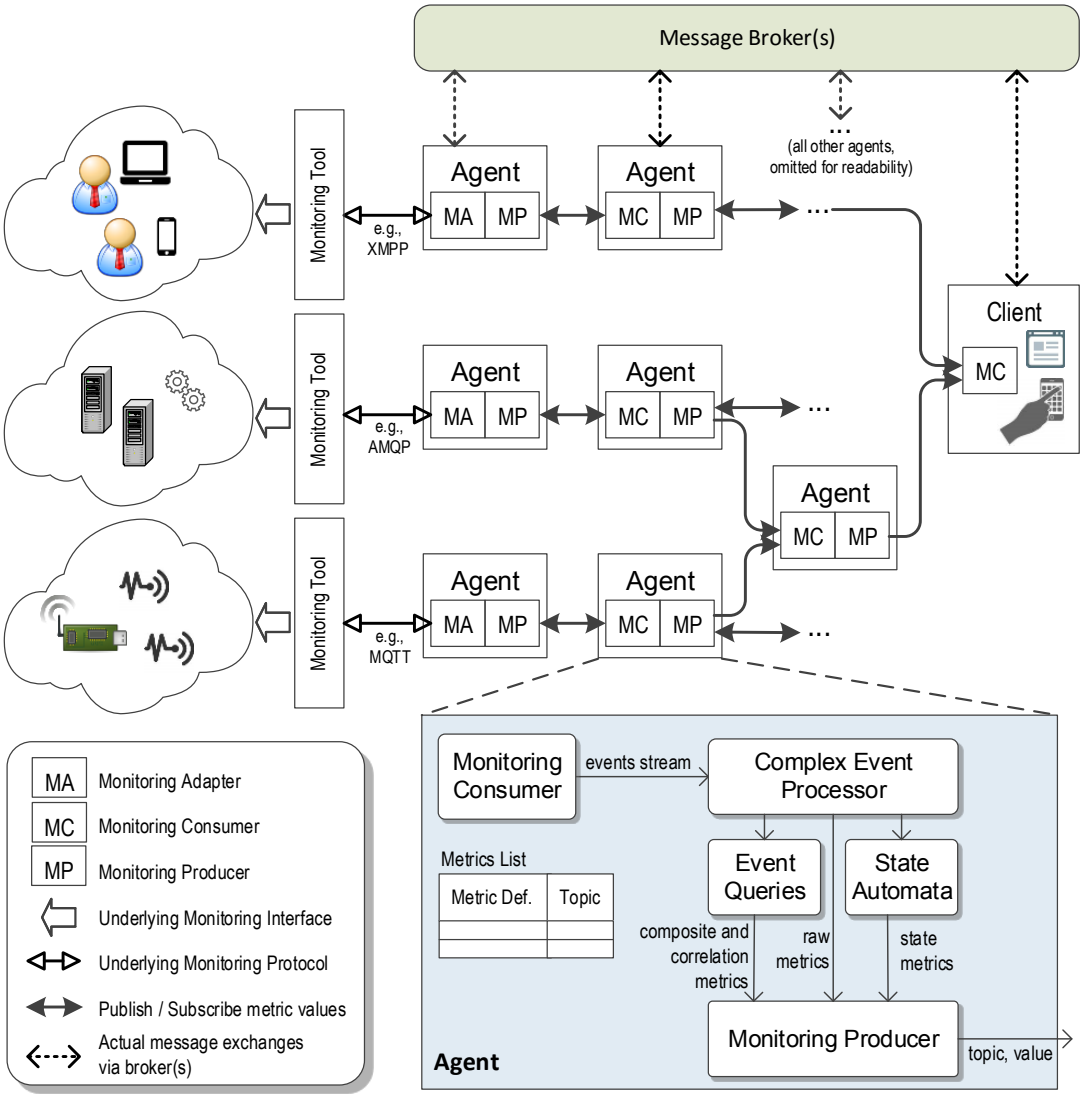


Figure 6.3: Monitoring Framework

can be utilized, e.g., aggregating an incoming event stream using an aggregate function, or combining multiple event streams using a composite expression, or normalizing and correlating multiple event streams. State automata can be employed to listen to state events from the stream and to produce state-based metrics. Each of these produced metric streams are then published by the MP under a specified topic. Proposed models for implementing QoD are discussed in the following subsection.

6.3.2 Protocol and Quality-Aware Delivery

There are currently a multitude of protocols supporting the publish/subscribe pattern. Our proposed framework focuses on the abstraction for dealing with monitoring entities involved in an HCS. Hence, the realization of such an abstraction may use available protocols.

A typical implementation of such publish/subscribe protocols decouples publishers and subscribers by employing a message broker (or a cluster of brokers), which has the logic for routing message exchanges between publishers and subscribers. To the best of our knowledge, currently there are no protocols that have out-of-the-box support for a dynamic message exchange routing which allows one topic to be delivered to multiple subscribers having different quality requirements with respect to the data rate, accuracy, or freshness. However, the implementation of the QoD-aware data delivery may extend available subscription message format, when possible; and then use the custom exchange routing to implement the QoD-aware data delivery algorithm.

The implementation of such QoD-aware data delivery can be done on two sides, i.e on the broker side or on the client (agent) side. Implementing QoD-aware delivery on the broker side is only possible for protocol that supports custom exchange routing, for example on an AMQP-based implementation (e.g., RabbitMQ). For a protocol that does not allow a custom exchange routing, e.g., MQTT, the QoD-aware delivery implementation is only feasible on the agent side.

In agent-side QoD-aware delivery, the publisher must know which subscribers are listening to topics with the required QoD, so that the publisher knows exactly to whom and when messages should be sent. Hence, the agent-side QoD-aware delivery breaks one of the original goals of the publish/subscribe pattern, i.e., the decoupling of publishers and subscribers. Moreover, the broker-side QoD-aware delivery puts all the QoD processing logic on the broker, hence making the implementation of agents simpler. However, the agent-side QoD-aware delivery allows more optimized metrics publication, because it allows more granular control on when a publisher should publish a metric, instead of publishing on every produced metric values.

6.4 Reasoning for Adaptation

The purposes of a monitoring framework are manifold. For example, a monitoring framework is useful to provide inputs for ad-hoc analytics and decision supports to obtain meaningful insights, e.g., [158], or to provide metrics for operational dashboard, e.g., [159].

Another useful and widespread use of a monitoring framework is in autonomic computing. In autonomic systems with intelligent *monitor-analyze-plan-execute* control loop, a monitoring framework delivers the *monitor function*, which provides the mechanisms that collect, aggregate, filter and report details, e.g., metrics, collected from a managed resource [160].

To enable autonomic adaptation in HCSs, e.g., for adapting the composition of a running compute units collective (see Section 5.5), an adaptation engine can be developed using a reasoning engine, and implement the aforementioned monitoring consumer interface to receive metrics from our monitoring framework. Once the metrics of concern are defined, a set of rules can be defined to reason about actions to be performed when certain conditions are occurred.

For this adaptation reasoning to function, metrics need to be augmented with metadata, which represent the objects that own the metrics, e.g., task id, compute unit id, etc. During the lifecycle of the tasks, the adaptation engine continuously asserts facts representing metrics and their metadata to the reasoning engine according to the incoming events and metrics streams subscribed by the monitoring consumer. When a certain rule condition is met, the reasoning engine executes the adaptation actions.

Adaptation rules can be used to make decisions and invoke actions exposed by the compute units manager and the runtime environment. For example, an adaptation rule may decide to invoke an escalated action when a service level objective (SLO) is violated. Some rules may also be employed to apply changes on the property of the running compute units collectives due to certain conditions, such as imposing rewards and punishments to assigned human-based compute units. Furthermore, adaptation rules may also decide to adapt the formation of the running compute units collective, such as scaling-out (e.g., adding more experts to troubleshoot an issue), scaling-in (e.g., reducing the number of sensors), or swapping a compute unit with another one (e.g., replacing a malicious worker), etc. Such formation adaptation can be executed by making a re-provisioning request as described in Section 5.5.

The implementation of an adaptation engine varies depending on the technologies, e.g., the reasoning engine, that are being used. An example prototype implementation of an adaptation engine using Drools rule engine is presented in Chapter 4. In such implementation, the monitoring consumer subscribed to metric topics required by the consumer-defined rules written in Drools language. During runtime, for each retrieved metric, a *Metric* object containing the metric's value, topic name, and metadata is sent to the Drools' fact-base. A collection of useful actions, e.g., for invoking remote Web services provided by the compute units manager and the runtime environment, are collected in *Action* class. Listing 6.1 shows an example of a rule for replacing human-based compute units that has more than 12 hours of continuous activity. The *HumanActiveDuration* topic represents a utilization metric of a human-based compute unit as defined in Listing 6.2 in Section 6.5.

```
1 /* @subsribeTo: HumanActiveDuration */
2 rule "ReplaceHumanUnit"
3   when
4     metric : Metric(topic="HumanActiveDuration", value > 12*3600, unit.type="human")
5   then
6     Action.invokeService("/reprovision", metric.collective.id, metric.unit.id);
7 end
```

Listing 6.1: An example of Drools rule for $MaxUtilViolated$

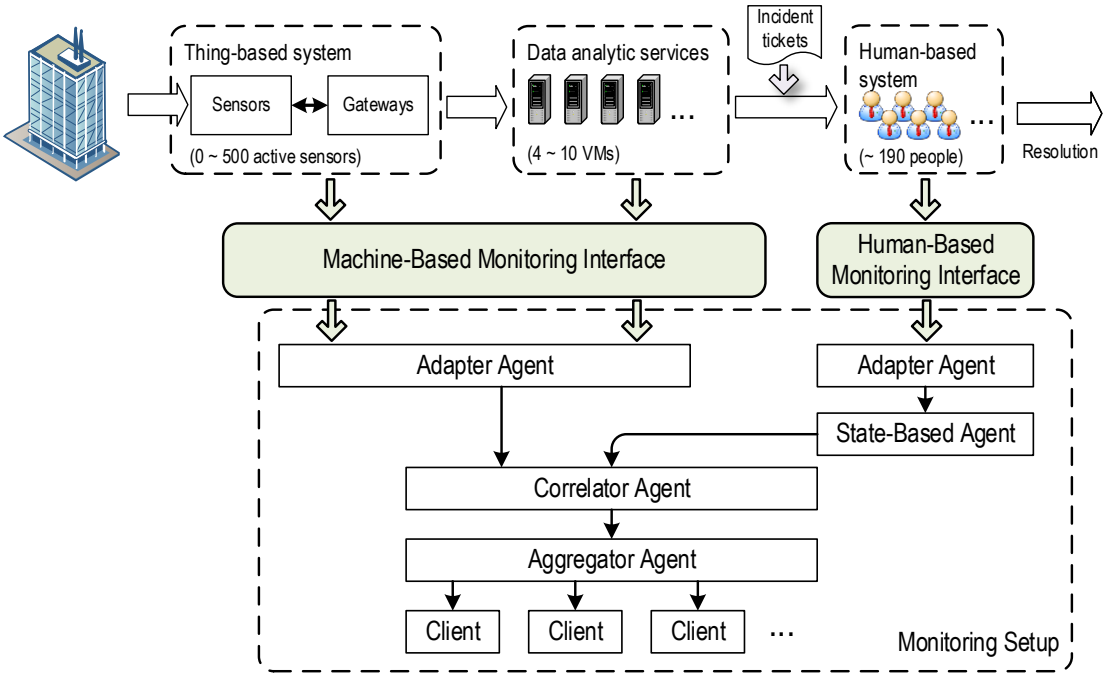


Figure 6.4: Monitoring Experiments Setup

6.5 Evaluation

To exemplify and evaluate our approach, we run experiments using our prototype platform presented in Chapter 4. In our platform, we employ Esper[1] as the complex event processors, and the evaluated metrics are translated into Esper event processing language (EPL). For the QoD processing, we implemented both, broker-based, and agent-based approaches.

We run experiments based on the infrastructure maintenance scenario discussed in Section 1.2. However, as shown in Fig.6.4, here we focus on the monitored metrics for sensors, which emit sensor data from an infrastructure, data analytic services, which analyze streams of sensor data, and dedicated human experts, who handle occurring incident tickets.

Experiments Setup To demonstrate the diversity of the underlying systems we monitor, we setup experiments employing monitoring data from a thing-based and software-based system, as well as a human-based system. We use data from a realistic Machine-to-Machine (M2M) DaaS, which processes information originating from several different types of data sensors (e.g., temperature, atmospheric pressure, or pollution). This M2M DaaS is comprised of processing and data services executed using an elastic cloud service framework, ADVISE [161]. The datasets of this experiment are available

[1]http://esper.codehaus.org

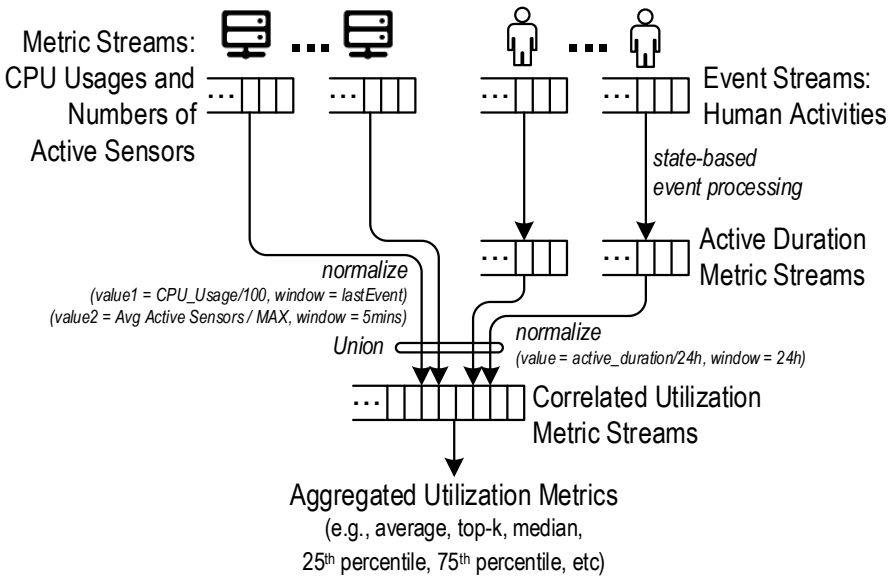


Figure 6.5: An Example of Processing and Correlating Streams

online[2]. During the execution we injected events that create incidents, which should be further investigated by human-based compute units. Based on a real incident management system, we simulate the composition and execution of the so-called *social compute unit* (SCU), which contains a group of experts that can be composed and dissolved on-demand [144]. The proprietary dataset of this incident management system is obtained from our industry partner, which contains a distribution of the human tasks occurrence according to real historical data, as well as the composition of the group of people performing each task, and the duration of the task execution.

For evaluation purposes, we created adapters for the underlying monitoring tools capable of retrieving and replaying the recorded monitoring data from the aforementioned setup. We implemented generic classes of monitoring agents, namely state-based agent, correlator agent, aggregator agent, and client agent. Together with a messaging broker, these agents are then incorporated as grid entities in our platform using GridSim framework (see Section4.1).

Experiment 1 - Retrieving Complex Metrics Traditional monitoring systems typically deal with homogeneous systems, where correlating similar metrics with different semantics from different subsystems is difficult. In our first experiment, we demonstrate the capability of our framework for capturing complex metrics derived from the correlation of metrics of different subsystems. Here we use the *utilization* metrics as discussed in Section 6.2.1. The utilization of human-based compute units is derived from their active hours during the last 24 hours. The utilization of the software-based system is obtained

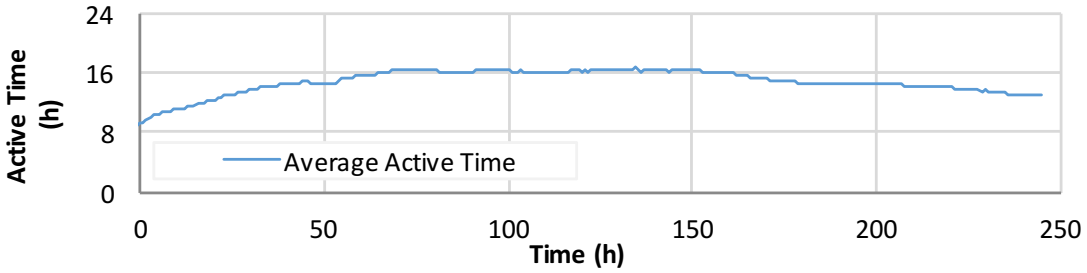
[2]https://github.com/tuwiendsg/ADVISE/tree/master/data/M2MApp

from the CPU usages of the machines running the software-based services. On the thing-based system, we capture the snapshots of the numbers of active sensors at any particular time. These three different metrics are then correlated, i.e., normalized and combined into one metric stream, so that further unified operations becomes possible. We then applied stream data aggregation operation (median and percentiles) to obtain new aggregated utilization metrics, which represent the behavior of the overall system.

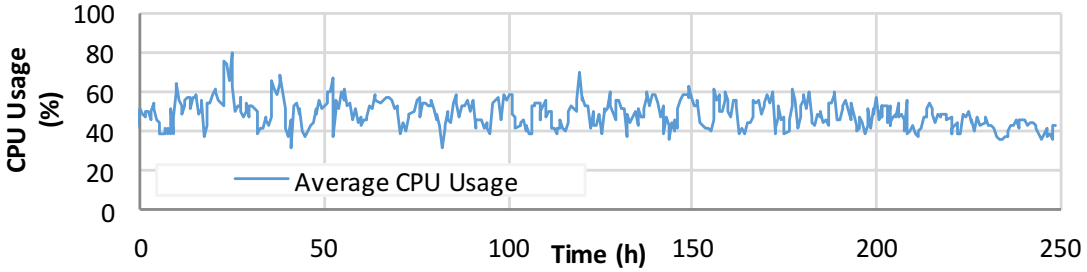
```
<metric name="HumanActivity">
  <type>raw</type>
  <interface>http://example.com/human/activity</interface>
</metric>
<metric name="HumanActiveDuration">
  <type>state</type>
  <transitions>
    [state transitions, omitted for brevity]
  </transitions>
  <expression>duration(STATE_ACTIVE)</expression>
</metric>
<metric name="CPUUsage">
  <type>raw</type>
  <interface>http://example.com/machine/cpu_usage
  </interface>
</metric>
<metric name="NumberOfActiveSensors">
  <type>raw</type>
  <interface>http://example.com/sensor/active_count</interface>
</metric>
<metric name="CorrelatedUtilization">
  <type>composite</type>
  <union>
    <source>
      <metric>HumanActiveDuration</metric>
      <window>24 hours</window>
      <normalize>value / 86400</normalize>
    </source>
    <source>
      <metric>CPUUsage</metric>
      <window>1</window>
      <normalize>value / 100</normalize>
    </source>
    <source>
      <metric>NumberOfActiveSensors</metric>
      <window>5 minutes</window>
      <normalize>avg(value) / MAX_SENSORS</normalize>
    </source>
  </union>
</metric>
<metric name="MedianOfSystemUtilization">
  <type>composite</type>
  <expression>median(CorrelatedUtilization)</expression>
</metric>
```

Listing 6.2: XML Definition for Correlating Utilization Metrics

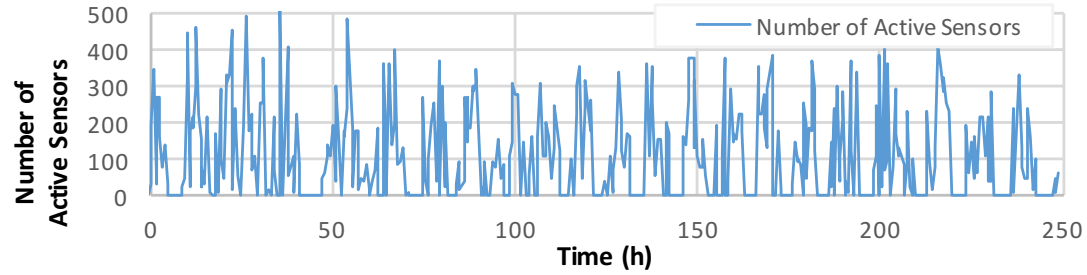
An XML definition of such correlated utilization metric is shown in Listing 6.2. Such a metric definition is then transformed into EPL and deployed into a complex event processor. Fig. 6.5 illustrates how the metrics and events streams are processed by the agents. We deploy the metric processors into our prototype implementation running the



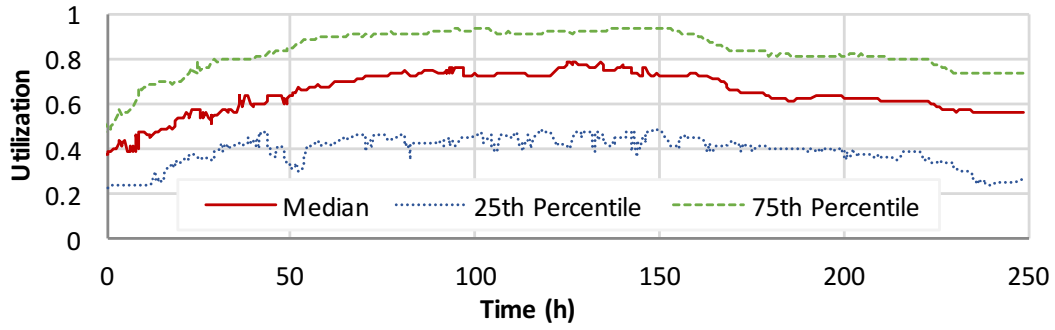
(a) Human-Based Units Utilization (average active time in the last 24 hours)



(b) Software-Based System Utilization (average CPU usages on all data collection and processing machines)



(c) Thing-Based System Utilization (the number of active sensors)



(d) Hybrid Human-Machine Computing System Utilization

Figure 6.6: Correlated Utilization Metrics

aforementioned infrastructure maintenance scenario and capture the resulted metrics as shown in Fig. 6.6. The streams of CPU usage and active sensors' metrics are fluctuated much rapidly as shown in Fig. 6.6b and Fig. 6.6c, while the active time of human-based compute units are more steady (Fig. 6.6a). We remove the data captured from the first 24 hours to avoid the effect of incomplete initial collection of human-based compute units activities. The outcome of the correlated utilization metrics shown in Fig. 6.6d.

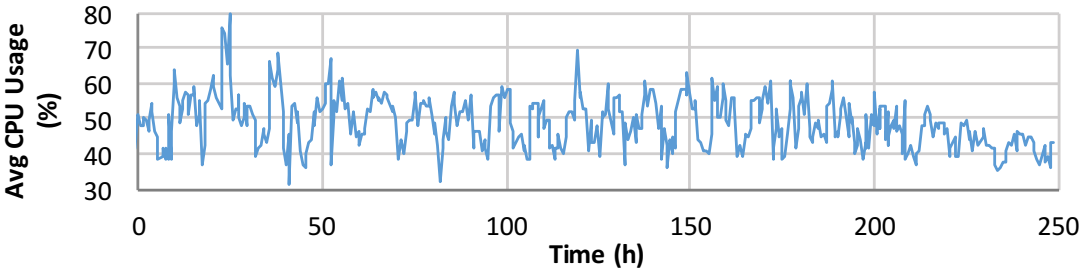
Experiment 2 - Non-intrusive Monitoring using QoD In HCS monitoring, different monitoring clients may require different data qualities. In this experiment, we would like to show the benefits of QoD-aware data delivery provided by our framework, especially for the monitoring clients with respect to the intrusiveness of the data. We deploy two monitoring clients that subscribe for CPU usage metrics. The first client subscribes without QoD requirements, while the second one emulates a human-based client, who wants only to receive updates on every 12 hours, while still requiring data accuracy of 10 points.

Here we use again a similar setup as in the first experiment, and apply the algorithm for QoD-aware data delivery shown in Algorithm 6.1 on the message broker. The estimation of the QoD-aware data is using moving average to calculate the data value on a particular point. As can be seen in Fig. 6.7, the data received by the second client is much more sparse than the first one, as it requests to receive data on every 12 hours basis. However, on the events where the metric fluctuates very rapidly (i.e., more than the requested 10 points before the 12 hours duration dues), the clients receives more data.

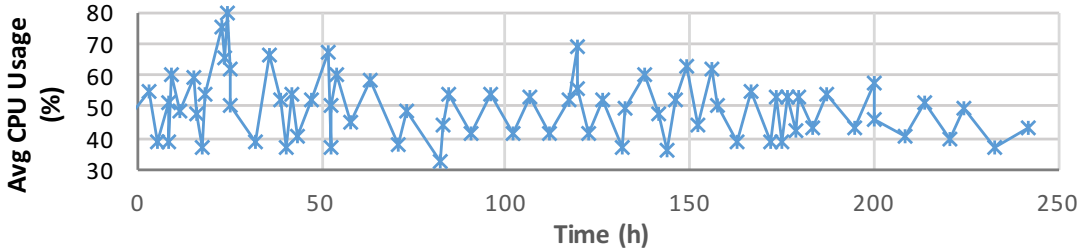
Experiment 3 - Comparing Implementations of QoD-aware Data Delivery As discussed in Section 6.2.2, QoD-aware data delivery can be implemented either on the broker-side or on the agent-side. In this experiment, we want to compare both approaches, and study the costs and benefits, especially from the perspective of monitoring providers. Here we experiment using similar setup as in Experiment 1, and apply the QoD-aware data delivery algorithm on either the broker or the agents and evaluate the results based on the number of messages, which represent the monitoring overhead for the overall system. The messages are counted and classified in two classes, the published messages (i.e., messages sent out by agents to the broker), and fan-out messages (i.e., message sent out by the broker to consumers).

First, we run the experiments using varying number of clients, i.e., 20, 40, and 60 clients, each with varying QoD requirements. As shown in Fig. 6.9, the broker-based quality-aware delivery is more efficient compared to the agent-based counterpart with respect to the number of total messages. This is due to the fact that the number of published messages on the broker-based quality-aware delivery is constant regardless the number of clients; while on the agent-based quality-aware delivery, the published messages are addressed to each clients with particular QoD requirements.

However, the agent-based quality-aware delivery can be more efficient than the broker-based one on different setups. Here, we setup again the experiments with 10



(a) CPU Usages without QoD



(b) CPU Usages with QoD (rate = 12h, accuracy = 10.0)

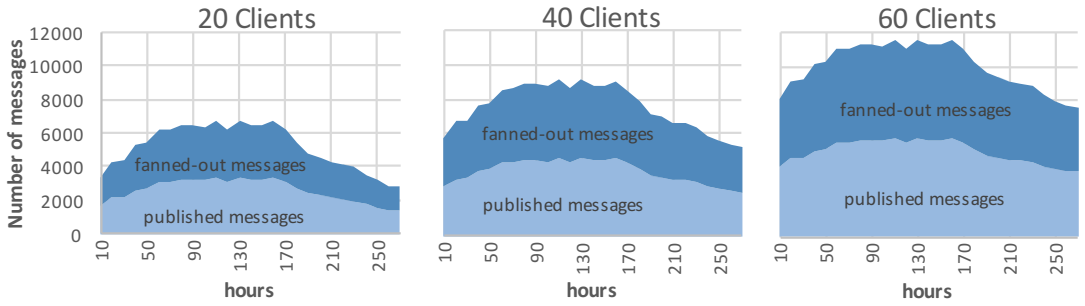
Figure 6.7: Quality of Data (QoD) Experiments

clients. We run several sets of experiments with these clients, each set with different data rates requirements as shown in Fig. 6.9. Here we can see that the agent-based quality-aware delivery is more efficient in low data rate requirements, because the number of its published messages becomes lower than the number of published messages in the broker-based counterparts. The cross points of these two approaches represent the data rates that are roughly equal to the mean original data rate (i.e., the data rate of messages sent out by agents if there is no QoD requirements).

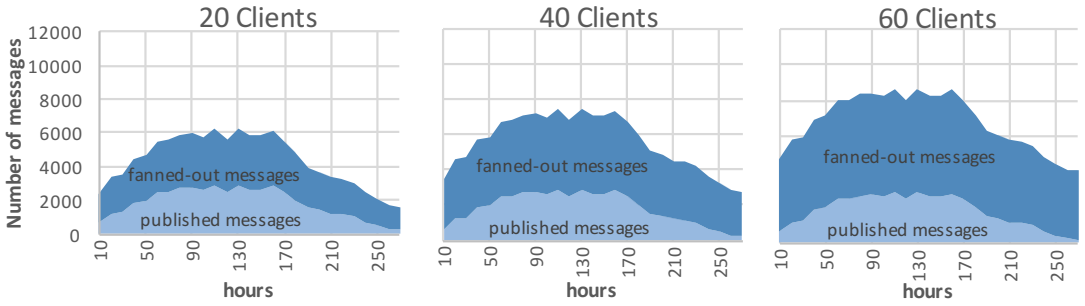
6.6 Chapter Summary

In this chapter, we presented our framework for monitoring an HCS considering the characteristics of the underlying subsystems. We proposed four classes of metrics to model both simple metrics, and complex metrics. Especially, the construct of correlation metric is used to tackle the problem of combining together metrics with different semantics from diverse compute units. Moreover, we applied the concept of Quality of Data (QoD) to cater custom monitoring requirements, which represent a trade-off between data quality and monitoring cost.

Our experiments showed the effectiveness of our monitoring framework to capture complex metrics. Furthermore, we showed benefits for both, monitoring clients, and providers, in applying QoD-aware data delivery on HCS monitoring.



(a) Agent-based Quality-aware Delivery



(b) Broker-based Quality-aware Delivery

Figure 6.8: Number of Messages in Quality-Aware Delivery

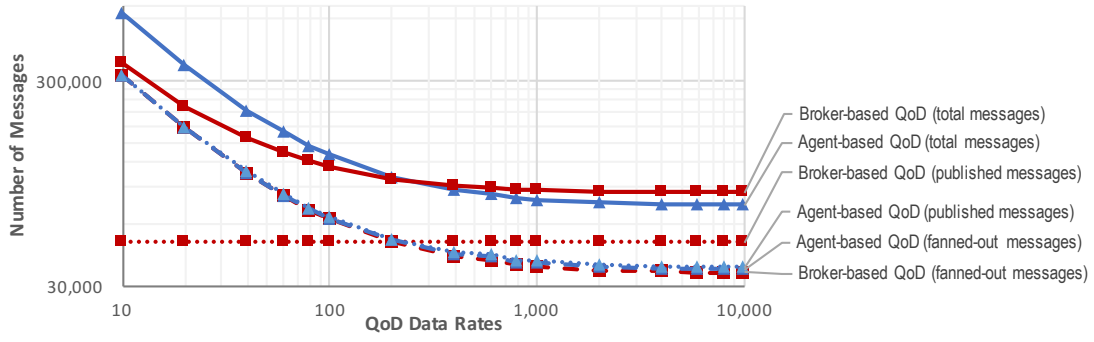


Figure 6.9: Number of Messages in Varying Data Rates

CHAPTER 7

Reliability Analysis

7.1 Introduction

Reliability is one of the important quality measures of a system. In a traditional machine-only computation, *reliability* is typically defined as the ability of a system to function correctly over a specified period, mostly under predefined conditions [162]. However, in the context where human-based compute units are involved, i.e., in Hybrid Human-Machine Computing Systems (HCSs), the *reliability* property is used with different quantifications and interpretations, e.g., the reliability property can be interpreted as (i) the probability of human errors so that such errors can be mitigated to obtain a high level of safety environment [134, 135], e.g., in healthcare, and transportation sector, (ii) the ratio of successful task executions in a workflow or a business process, e.g., [132, 133], or (iii) the quality of results or contents, e.g., [36, 163, 131].

In HCSs, we are interested in understanding the reliability of the provisioned compute units collectives to execute tasks. However, analyzing the reliability of compute units collectives introduces many challenges. The diversity of the compute units and their individual reliability models brings forth different failure characteristics that must be taken into account when measuring the reliability. The complexity of the structure of the compute units collective and the large scale of the involved compute units also contribute to the complexity of the reliability analysis.

Our contribution presented in this chapter answers *Research Question 3: "How to measure the reliability of an HCS, which consists not only machine-based compute units but also human-based compute units?"*. The salient contributions of this chapter is to propose a framework for compute units collectives reliability analysis, which takes into account individual compute units' reliability model and the compute units collective's structure. We adopt models to measure the reliability of individual machine-based and human-based compute units, and introduce a model that can be used for describing the complex structure of compute units collectives, i.e., the *collective dependency* model discussed in Section 3.3.4. Furthermore, to deal with the large scale of the cloud-based

landscape, we introduce the notion of *virtual standby units* abstracting the group of compute units available from the pool of computing resources. These models are then utilized to perform the reliability analysis.

A set of tools for modeling and analyzing the reliability of compute units collectives is useful, e.g., (i) for application designers to design, evaluate, and improve application components for executing tasks, (ii) for resource platform providers to deliver more reliable machine-based and human-based compute units such as by providing a reliability-aware discovery and composition service, and (iii) for task owners to tune the task specification to achieve the required reliability.

7.2 Reliability Models

In machine-based computation, failures are typically caused by natural- or design-faults [162]. However, for human compute units the nature of the faults is different. Humans are prone to execution error [134]. When a human performs a task, it is natural he/she performs an error, which leads to failure. Also, same tasks executed by the same worker on different times may give different results.

In general, reliability models can be categorized into black box and white box models [162]. For human-based compute units, it is complex to model the internal functioning of a human work using a white box model. Black box models, such as based on interpolation or parameter estimation using historical data, can be used for predicting the individual reliability. Various influencing factors, such as trust, skills, connectedness of the compute units collectives, as well as past success rates, may affect the reliability of individuals. However, problems may arise for a new compute unit with no historical data. To this issue we point to approaches for predicting reliability based on similarity such as found in [164]. Our work presented in this chapter focuses on the issues of the reliability analysis for mix human-machine collectives using black-box models with *a priori* known factors.

7.2.1 Reliability of Individual Units

Reliability of machine-based compute units Measuring the reliability of machine-based compute units is a well-researched problem [165, 166, 167]. Generally, it can be summarized as follows. Let T be a continuous random variable that represents the time elapsed until the first failure occurs. And let $f(t)$ be the probability density function of T, and $F(t)$ be the cumulative distribution function of T. Traditionally, $F(t)$ represents the unreliability of the system, i.e., the probability that the system fails at least once in time interval $[0, t]$. The reliability, $R(t)$, of the compute unit is the complement of $F(t)$, i.e., $R(t) = 1 - F(t)$ [165].

Reliability of human-based compute units In human-based tasks, we typically do not deal with the exact time when a particular human-based compute units fails, instead we are more interested in whether a particular task execution is likely successful.

Furthermore, in the execution of human-based tasks, the active execution time of the human-based compute units is not continuous, i.e., people may take a break, eat, and sleep. Therefore, in our model, we approach the reliability of human-based compute units using a discrete time space.

Let K be a discrete random variable which represents the number of consecutive successful task executions by a particular human-based compute units until a first failure occurs. Let $f(k)$ be the probability density function of K which also represents the probability of the first failure occurs at k-th task execution. Let $F(k)$ be the cumulative distribution function of K. $F(k)$ represents the unreliability of the human-based compute unit, i.e., the probability that the compute unit fails at least once in execution $[1, k]$. The reliability, $R(k)$, defines the reliability of the human-based compute units for the execution of all k tasks. Hence, we have

$$\begin{aligned} f(k) &= Pr(K = k) \\ &= Pr\{task_k \text{ fails } \mid task_1, task_2, ...task_{k-1} \text{ succeed}\}. \end{aligned} \tag{7.1}$$

Depending on the problem domain and the underlying human-computing systems, different discrete distributions can then be employed to define $f(k)$. Note that the distribution parameters of such failure probability may also dynamically change from time to time, e.g., due to the human skill evolution [168]. To exemplify this model, in our experiments described in Section 7.4, we approach $f(k)$ using a geometric distribution with non-dynamic parameters.

This model extends models proposed in human reliability analysis and task quality measurement techniques, e.g., [131, 132, 133, 134, 135], where the reliability property, e.g., with respect to the failure/success probability, can be taken for granted. However, instead of using only a single value of failure/success probability for the next human task execution, our model allows the estimation of the reliability as a cumulative probability of failure/success within a set of consecutive task executions. Hence, together with the traditional reliability measurement of machine-based compute units we could derive the reliability of compute units collectives in a discrete time space.

7.2.2 Reliability of Compute Units Collective

We define the reliability of a compute units collective as the reliability of the task execution performed by the compute units collective, i.e., the probability that the compute units collective successfully execute tasks. As discussed in the following section, the reliability of a compute units collective to execute a task depends on the reliability of the individual compute units that are potentially assigned and the structure of the compute units collective represented by the collective dependency.

7.3 Reliability Analysis Framework

Here we present a framework that provides features to evaluate the reliability of compute units collectives. The goal of our framework is to measure the reliability of the HCS

consisting compute units collective instances to execute tasks. More specifically, given a set of k consecutive tasks $T = \{t_1, t_2, ..., t_k\}$, we measure the reliability of all compute units collectives provisioned by the HCS to execute $t_i \in T$. This framework takes the following as inputs: the profiles of the compute units [36] to determine their individual reliability (Section 7.2.1), and the collective dependency of the task type (Section 3.5). Our proposed analysis approach yields the reliability of compute units collectives provided by the HCS to execute a particular task type.

7.3.1 Reliability Structure of Compute Units Collective

We revisit the compute units collective provisioning view shown in Fig. 3.3, Section3.2, where a compute units collective structure is made up of roles fulfilled by selected compute units according to the task requirement. The member compute units of a role can be fulfilled either (i) from a *static set of compute units* (e.g., as in Role Stream Analyzer, Role Human Computing Platform, Role Infrastructure Manager, and Role Communication Provider shown in Fig. 3.5) or (ii) from a *dynamic set of compute units* (e.g., as in Role Collector, Role Assessor, and Role Sensors also shown in Fig. 3.5).

For the dynamic set of compute units, one of the main challenges in a cloud-based compute units collective provisioning is to deal with large numbers of available compute units. For example, the number of people participating in a crowdsourcing platform can be very large (e.g., in a smart city). However, since compute units collectives are task-oriented and provisioned on-demand, we can abstract these large pools of compute units that will likely be included in the assembled compute units collective. We call these sets of compute units as *Virtual Standby Units* (VSUs). Hence, a VSU is a subset of the compute units pool consisting compute units qualified to perform a particular role. A VSU consists of a set of active compute units assigned to execute the task, and a pool of qualified standby compute units, as shown in Fig. 7.1. During runtime, a detection-and-reconfiguration component monitor the active compute units to detect any failure, and when a failure occur, the active compute units can be reconfigured by replacing the faulty compute unit with the one from the pool of standby compute units.

With this approach, only compute units qualified for providing resources required for the task's roles are considered for analysis. However, the construction of VSUs should consider not only static profiles, but also the dynamic changes of functional and non-functional properties of the compute units. For example, in our infrastructure maintenance scenario, we may want to analyze the reliability of the facility sensing capability against a particular building at a particular time slot; therefore, we can utilize the participants properties such as time availability and location history to decide whether he/she should be included in the VSU for that particular task.

Given a set of all compute units $\mathcal{U}$, the VSU for t is a subset of $\mathcal{U}$, i.e., $VSU_t \subset \mathcal{U}$. The members of VSU_t can be retrieved using the discovery service provided by the compute units manager considering the properties of the compute units (see Section 3.1.2).

On the task level, the reliability structure of a compute units collective depends on its collective dependency model, which has been presented in Section 3.3.4). For the shake of readability, in Fig. 7.2, we show again the collective dependency model of

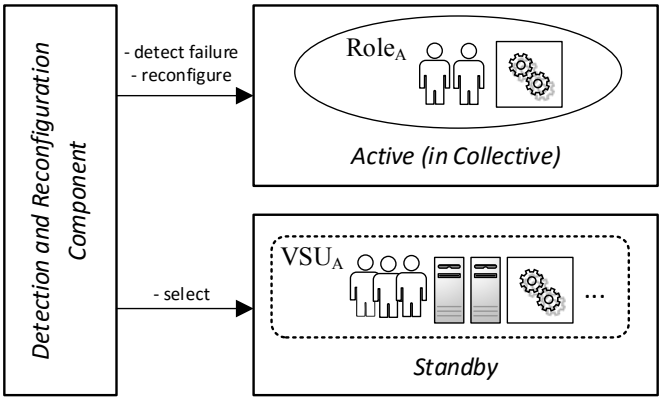


Figure 7.1: VSU's Structure

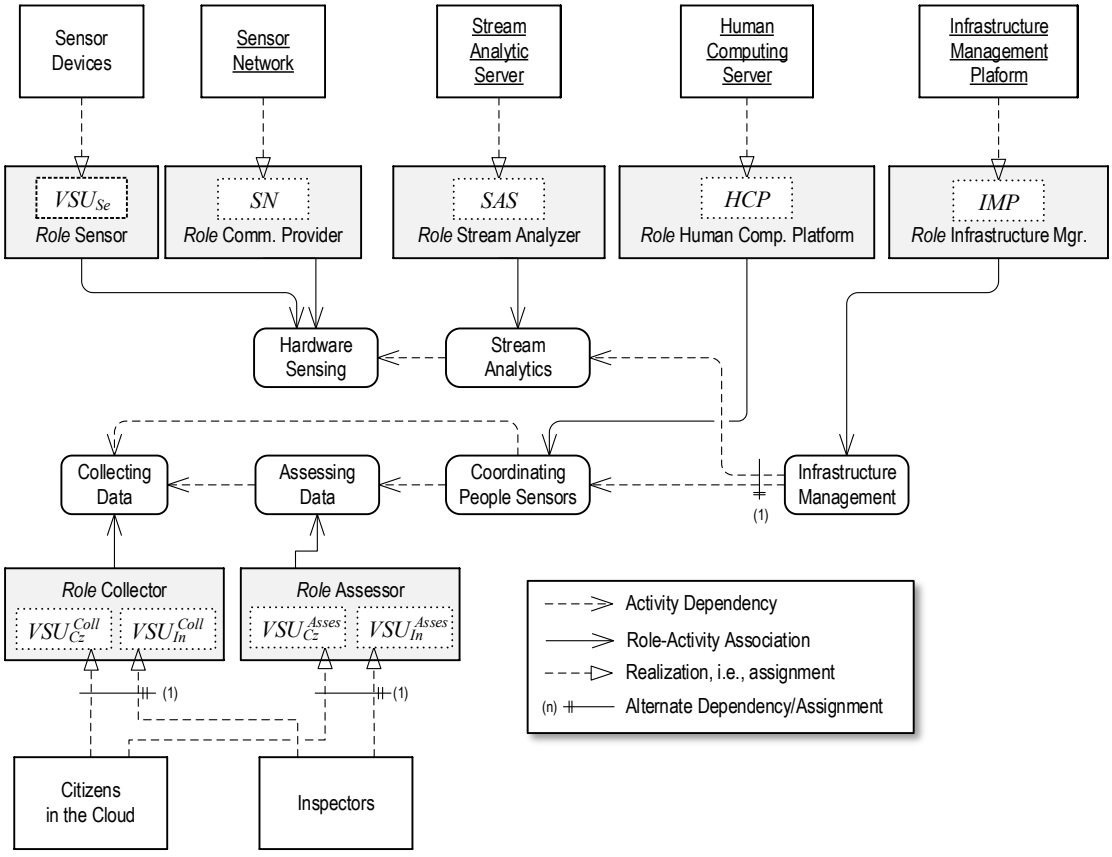


Figure 7.2: Collective Dependency for Reliability Structure

our infrastructure scenario, previously shown in Fig. 3.5, with additional labeling for the static set of units and the VSUs for each roles. Here, the sensors (Se), citizens (Cz), and inspectors (In) are constituted as VSUs, and both citizens and inspectors may provide services for collector ($Coll$) and assessor ($Asses$) roles, hence we have the following VSUs: VSU_{Se}, VSU_{Cz}^{Coll}, VSU_{Cz}^{Asses}, VSU_{In}^{Coll}, VSU_{In}^{Asses}. Furthermore, the Infrastructure Management Platform (IMP), the Stream Analytic Server (SAS), the Human-based Computing Platform (HCP), and Sensors Network (SN) are static sets of compute units.

7.3.2 Reliability Calculation

We employ the following procedures to estimate the reliability of compute units collectives:

1. We calculate the reliability of individual compute units based on their profiles.
2. We determine the reliability for each group of compute units potentially assigned for a particular role.
3. We calculate the reliability of the executions of task instances for a particular task type based on the reliability of the group of compute units assigned for each task roles.

The first procedure has been discussed in Section 7.2.1. In the following we discuss the last two procedures in detail.

Reliability of Role Assignments

The reliability for each role assignments in a compute units collective is defined according to the reliability of the set of compute units that can be assigned for the role. Compute units assigned to a particular role can be either from a static set of compute units, or from a VSU. We discuss the reliability of the sets of compute units according to these two types of assignments as follows.

a) Reliability of static sets of compute units A static set of compute units may employ only a single compute unit (simplex), or a certain basic structure such as the parallel structure, where we distribute a task in parallel and expect at least one compute unit returns a result, or the series structure, where we expect all assigned compute units provide results correctly. A more complex structure can also be formed from these basic structures. A static set of compute units may also employ a *static redundancy* for masking faults. One of well-known approaches for a static redundancy is the *M-of-N* redundancy, which consists of N compute units and requires at least M of them to function properly. For example, in human-based computing, this *M-of-N* redundancy can be in the form of assignments of the same task role to N people, where we expect at least M people provide the correct result reliably. The calculation of the reliability of such static structures is well known and can be found in [167].

b) Reliability of VSUs When a role of a task is fulfilled using potential compute units from an VSU, it resembles the structure of a set of active compute units accompanied by standby spare compute units, as shown in Fig. 7.1. If any of the active compute units fails, a standby compute unit is activated for a replacement. This resilience approach is traditionally called *hybrid redundancy* (or simply *dynamic redundancy* when only a single compute unit is active), where we dynamically detect (or predict) faults and reconfigure the structure of the running compute units collective to correct (or anticipate) the faults. In this case, we also need to take the reliability of the detection and reconfiguration component into account.

If the active compute units from an VSU are assembled to use *M-of-N* redundancy approach, we would need at least M compute units to function properly. Let L be the number of standby spare compute units, the reliability of the VSU is given by the probability that at least M compute units out of $L + N$ compute units are functioning correctly. Hence, given the reliability of the detection and reconfiguration component R_{DR} and the uniform reliability of each compute units R_u the reliability of an VSU is given by

$$R_{VSU} = R_{DR} \cdot \sum_{i=M}^{L+N} \binom{L+N}{i} R_u^i \left(1 - R_u\right)^{L+N-i}.$$

For non-uniform R_u, an analytical probability calculation based on each individual compute unit reliability can be performed.

Reliability of Task Executions

When a compute units collective is assembled, its assigned compute units constitute a configuration that fulfills a set of required dependencies as defined by the task's collective dependency model (see Section 3.3.4). Due to the flexibility of compute units collectives, i.e., defined by alternate dependencies and alternate assignments in the collective dependency model, different compute units collective configurations may be composed for different task instances. We use the concept of *execution spanning tree* (EST) to identify various possible compute units collective configurations for a particular task type.

We define that an EST contains the inter-dependent static sets of compute units and/or VSUs such that its vertices (the static sets of compute units/the VSUs) are capable to execute a set of required *c-activities* defined in the collective dependency model. That is, given a collective dependency graph $\mathcal{G}_{dep} = (\mathcal{A}, \mathcal{E})$ and static sets of compute units/VSUs $\mathcal{V}$, we can have an EST $\mathcal{T} = (\mathcal{V}', \mathcal{E}')$, where $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}'$ is the dependency of $\mathcal{V}'$ according to $\mathcal{E}$, such that $\mathcal{V}'$ and $\mathcal{E}'$ encompass one possible alternative dependency set in $\mathcal{G}_{dep}$.

To obtain ESTs, we could derive the dependencies between the compute units from the collective dependency. Algorithm 7.1 presents a procedure to transform a collective dependency into a set of ESTs. For example, given a collective dependency model shown in Fig. 7.2, we can obtain a set of possible ESTs as follows:

- IMP, SAS, VSU_{Se}, SN

- $IMP, HCP, VSU_{Cz}^{Coll}, VSU_{Cz}^{Asses}$
- $IMP, HCP, VSU_{Cz}^{Coll}, VSU_{In}^{Asses}$
- $IMP, HCP, VSU_{In}^{Coll}, VSU_{Cz}^{Asses}$
- $IMP, HCP, VSU_{In}^{Coll}, VSU_{In}^{Asses}$

For a compute units collective to execute a task reliably, at least one EST must successfully accomplish the task. The failures of all possible ESTs result to the failure of the compute units collective to execute the task. Therefore, given a task t and its set of EST $\mathcal{T}_t$, we can define the reliability to execute the task t, R^t, as the probably of having at least one EST of $\mathcal{T}^t$ working properly:

$$R^t = Pr\{\exists EST, EST \in \mathcal{T}^t \wedge EST \text{ works properly}\}.$$

Let E_i be the event that $EST_i \in \mathcal{T}^t$ operates properly, then the reliability to execute the task t is given by

$$R^t = Pr\left\{\bigcup_{i=1}^{|\mathcal{T}^t|} E_i\right\}. \tag{7.2}$$

The calculation of probability of such events should consider the fact that E_i may be correlated, i.e., the inclusion of VSUs in ESTs are not exclusive. Several works, e.g., [169, 138], propose some techniques to calculate such probability.

7.4 Evaluation

In the following, we apply our model by exemplifying some reliability analyses on different scenarios. Our goal here is to show how our model can be used to measure the reliability of task executions, R_{task}, and how we can get insights from the reliability analysis. In our experiments, we need to study the reliability of the systems with various different configurations with respect to the different numbers of compute units and their properties. Therefore, we use simulated pools of compute units, as well as simulated task requests. The purpose of this experiment is not to model a true-to-life scenario. However, we want to show how our tool can be used to model and tune different scenarios, and how the reliability analysis can be used as a feedback for improvements. Here, we use again our prototype platform presented in Chapter 4, to simulate various behaviors of an HCS and study their reliability.

In our experiments, we use the infrastructure maintenance scenario as depicted in Fig. 1.1 and Fig. 3.5. We define the task in our experiments as the task for sensing facility breakdown. Each instantiation of a task is implicitly associated with an occurring breakdown event. A task execution is said to be successful when the breakdown is correctly detected. Here, a reliability analysis is very important and useful for improving the reliability of the system. For example, we can identify which section of the city or building complex has unreliable sensing capability; therefore, we can schedule and

Algorithm 7.1: EST Generation Algorithm

```
1  Function generateEST(dependencyGraph)
2      ESTList ← ∅
3      foreach root ∈ dependencyGraph.getRoots() do
4          est ← generate(root)
5          ESTList ← combine(ESTList, est)
6      end
7      return ESTList
8  end
10
11 Function generate(node)
12     ESTList ← node.generateResourcesEST()
13     foreach branch ∈ node.getBranches() do
14         if branch.isAlternating() then
15             childESTList ← ∅
16             foreach altNode ∈ branch.getAltNodes() do
17                 alternateEST ← generate(altNode)
18                 childESTList.add(alternateEST)
19             end
20         else
21             branchNone ← branch.getNode()
22             childESTList ← generate(branchNone)
23         end
24         ESTList ← combine(ESTList, childESTList)
25     end
26     return ESTList
27 end
29
30 Function combine(list1, list2)
31     if list1 = ∅ then
32         return list2
33     else if list2 = ∅ then
34         return list2
35     else
36         resultList ← ∅
37         foreach s1 ∈ list1 do
38             foreach s2 ∈ list2 do
39                 resultList.add(s1 + s2)
40             end
41         end
42         return resultList
43     end
44 end
```

dispatch dedicated inspectors more frequently on that particular section, or recruit more citizens to increase the reliability of the sensing capability.

Without loss of generality, in these experiments we model the probability of failed executions of each individual compute unit in discrete time space using the geometric distribution. Let p be the failure probability of executions by an individual compute unit. Assuming that p is constant and independent of the execution time, we could have

$$\begin{array}{ll} f(k) & = (1-p)^{k-1}p \quad \text{, and} \\ F(k) & = 1-(1-p)^{k} \quad \text{, therefore} \\ R(k) & = (1-p)^{k}. \end{array} \tag{7.3}$$

An estimation of the distribution parameter p can be derived from the task execution data of each individual compute units. For a known compute unit u, let $e = e_1, e_2, ..., e_n$ be a set of result execution samples. The value of e_i may be a binary, 1 for a successful execution and 0 for a failure execution, or a floating number [0..1], which represents the result quality of the execution. The distribution parameter p of compute unit u can be estimated by

$$\hat{p_u} = 1 - \frac{\sum_{i=1}^{n} e_i}{n}. \tag{7.4}$$

Experiments Setup Assuming the Infrastructure Management Platform (IMP), the Stream Analytic Server (SAS), the Human-based Computing Platform (HCP), and Sensors Network (SN) are static sets of compute units (Fig. 3.5), we focus our experiments on studying the variability of VSUs configuration of machine-based sensors, as-well-as human citizens, and human inspectors in fulfilling *sensor*, *collector*, and *assessor* roles. Citizens and inspectors may be assigned to the collector and assessor role, while machine-based sensors are assigned to fulfill the sensor role. Each machine-based compute unit (i.e., a sensor) has a randomly generated continuous failure rate λ, and the reliability at a particular time t is given by $R(t) = e^{-\lambda t}$ [165]. Each human-based compute unit (i.e., a citizen or an inspector) has a randomly generated probability of failure p, and the reliability at a particular execution k can be measured using Equation 7.3. The way how the properties of compute units can be configured is discussed in Section 4.2 and Appendix A.2.

We perform three sets of experiments to study different aspects of reliability in compute units collectives with different configurations as shown in Table 7.1. These experiments are discussed as follows.

Experiment 1 - Reliability Changes over Time In this experiment, we study how the reliability of the task executions changes over time. We generate a fix number of compute units and generate statistically distributed failure probabilities and failure rates for each compute units as shown in Table 7.1. We employ fix reliability configurations: for citizens, when they are assigned to a task, at least 2 of 3 assigned citizens must be working properly; for inspectors and sensors, we require only one working compute unit. We simulate the detection and reconfiguration of faulty compute units using software-based

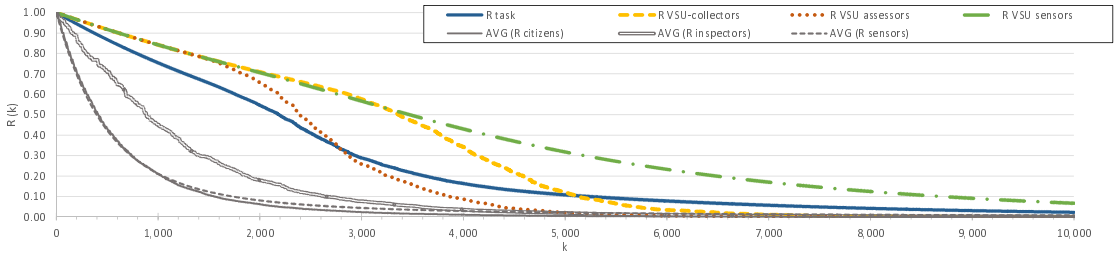


Figure 7.3: Reliability on task executions, $R(k)$

Scenarios	Goals: to study	Configurations	Variants
Exp. 1	reliability changes over time or executions	$N_{citizens} = 200$ $N_{inspectors} = 10$ $N_{sensors} = 50$ $\bar{p}_{citizens} = 0.3$ $\bar{p}_{inspectors} = 0.05$ $\bar{\lambda}_{sensors} = 0.02$ $\lambda_{DR} = 0.001$	$k = [1..10000]$
Exp. 2	effect of different sizes of compute unit pools on reliability	$\bar{p}_{citizens} = 0.3$ $\bar{p}_{inspectors} = 0.05$ $\bar{\lambda}_{sensors} = 0.02$ $\lambda_{DR} = 0.001$ $k = 2500$	$N_{citizens} = [0..300]$ $N_{inspectors} = [0..20]$ $N_{sensors} = [0..250]$
Exp. 3	effect of different compute units collective provisioning strategies on reliability	$N_{citizens} = 200$ $N_{inspectors} = 10$ $N_{sensors} = 50$ $\bar{p}_{citizens} = 0.3$ $\bar{p}_{inspectors} = 0.05$ $\bar{\lambda}_{sensors} = 0.02$ $\lambda_{DR} = 0.001$ $k = 2500$	strategies: - uniform distribution - fastest response - greedy (cost optimized)

Table 7.1: Reliability Analysis Experiment Scenarios

components with a failure rate $\lambda_{DR} = 0.001$. We generate 10,000 tasks with a task rate of 30 tasks per time unit. For each task instance, members of the VSUs are then assigned and activated for executing the task.

During the experiment we measure the average reliability of individual compute units, as well as the reliability of VSUs and the aggregated reliability of the task executions, R_{task} (given by equation 7.2), as shown in Fig. 7.3. The reliability of VSUs are affected by the number of compute units as well as the reliability of each compute units. $R_{VSUcollectors}$ is higher than $R_{VSUassessors}$ because in our experiments the number of generated compute units qualified for doing data collection task is around 50% more than the the number of data assessment qualified compute units. Furthermore, the average reliability of sensor compute units is calculated as a function of t; hence, the slope of its reliability is also affected by the task rate, here (as well as in other experiments) we use $t = \frac{k}{30}$, i.e., 30 tasks per time unit.

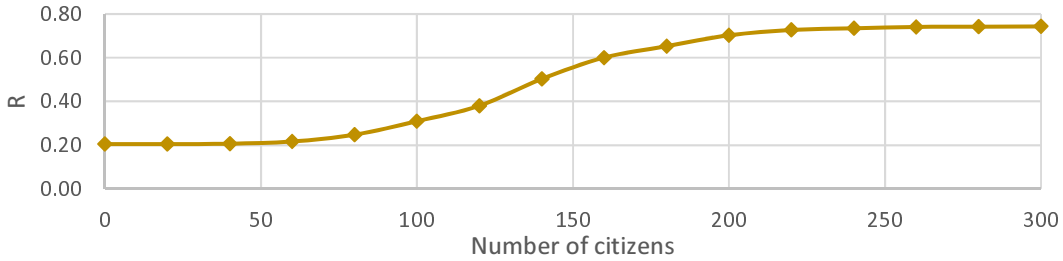
The reliability of VSUs are significantly higher than the average reliability of individual compute units, since VSUs employ dynamic/hybrid redundancy. The reliability of any computing systems always decreases over time. However, the decrement slope of an VSU is not as steep as its individual compute units. On low k value, the reliability of the whole system and VSUs are mainly affected by R_{DR}. In fact, in this setup if we simulate a perfect detection and reconfiguration component, i.e., $\lambda_{DR} = 0$, we will have $R_{task} \simeq 1$ until $k \approx 1000$. And R_{task} will drop below the average reliability of all individual compute units when $\lambda_{DR} > 0.0058$.

Therefore, to design reliable compute units collectives, we posit that the application designer should pay attention not only to the reliability of individual compute units, but also consider the structure of standby compute units, e.g., how they can be effectively discovered, and also the size of the available standby compute units. Furthermore, it is also important to design a highly reliable detection and reconfiguration component, otherwise the redundancy structure of the standby compute units will render useless.

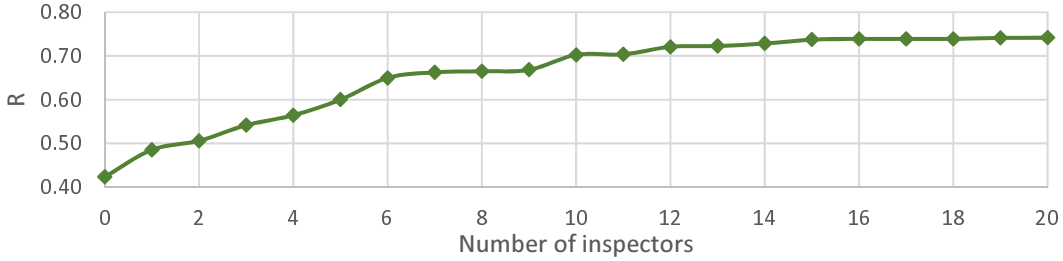
Experiment 2 - Effect of Different Sizes of Unit Pools Compute units in compute units collectives may come from different pools of compute units with varying quality and sizes. Our next experiments study how R_{task} is affected by the size of compute units pools. Such experiments may assist the resource platform providers to decide whether adding more resources is beneficial to improve the reliability of the compute units collectives.

We use the same values of p and λ, and employ the similar reliability configurations as the previous experiments. We experiment with 2500 generated tasks, i.e., $k = 2500$. Fig. 7.4 depicts how R_{task} changes with the varying number of citizens, inspectors, and sensors.

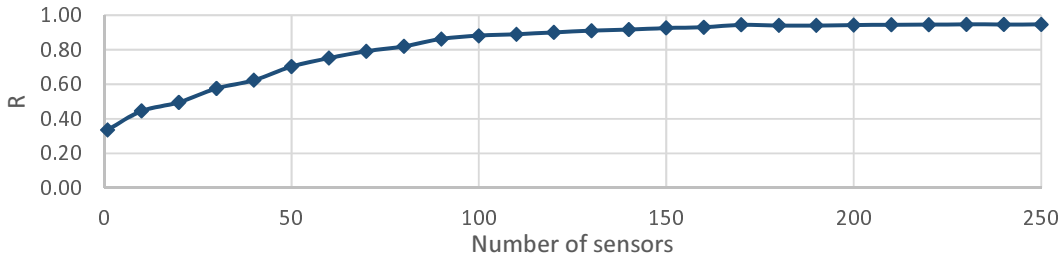
In these figures we can see that R_{task} values have upper limits due to the fact that other compute unit types (as well as other static components) are not being improved. By studying these R_{task}, we could recognize the sweet spots on which adding more compute units could effectively increase R_{task}. For example, the increment of the number of citizens between 80 to 220 on our setup effectively improve R_{task}, while adding more



(a) Reliability on varying number of citizens



(b) Reliability on varying number of inspectors



(c) Reliability on varying number of sensors

Figure 7.4: Reliability on varying size of resources pools

citizen compute units beyond 220 is fruitless. Hence, the importance of adding more compute units to increase the reliability (e.g., recruiting more citizens) must be balanced out with the recruitment cost.

As shown on Fig. 7.4a and Fig. 7.4b, the effect on R_{task} is greatly determined by the failure rate or failure probability. We require less additional recruitments of inspectors to improve the reliability due to $\bar{p}_{inspectors} \ll \bar{p}_{citizens}$. However, the structure of the corresponding VSUs also impacts the changes of R_{task}. In our setup, the role of the dedicated inspectors in $VSU_{collectors}$ and $VSU_{assessors}$ can also be replaced by citizens. Hence, we don't need many inspectors additions, compared to sensors additions, to improve R_{task}.

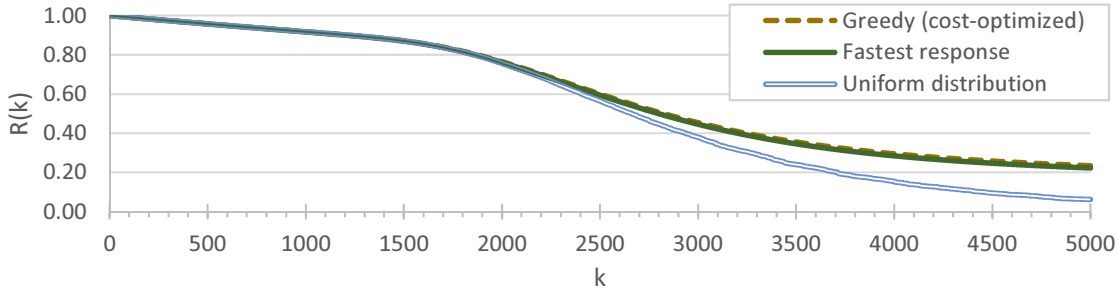


Figure 7.5: Reliability on different compute units collective provisioning strategies

Experiment 3 - Effect of Different Provisioning Strategies Different HCSs may employ different strategies for the provisioning of compute units collectives (see Section 5.3), which eventually affect the reliability of the compute units collectives as well as their non-functional properties. Here we experiment with three provisioning strategies: (i) the *uniform distribution* strategy, where the tasks are uniformly assigned to qualified compute units, (ii) the *fastest response* strategy, where the tasks are assigned to the qualified compute units that provide the fastest response times (e.g., depending on the compute unit's job queue and performance rating), and (iii) the *greedy* strategy, where we employ a greedy optimization algorithm, see Section 5.4, to minimize the execution cost of the compute units collective. The performance rating and the execution cost of each individual compute unit are statistically distributed during compute units generation based on the generator configurations (refer to Section 4.2 and Appendix A.2 for the simulation configuration).

We use similar configurations as in the first experiment with 5000 tasks. As we can see on Fig. 7.5, the reliability of the *fastest response* strategy and the *greedy (cost-optimized)* strategy are similar. However, we observe that the reliability of the *uniform distribution* strategy is lower than the other two, especially on higher k. This is due to the fact that the *fastest response* strategy and the *greedy (cost-optimized)* strategy tend to select a particular set of compute units with better performance rating and cheaper cost, respectively; hence, they yield more standby compute units with less utilization. On the individual level, a compute unit with less utilization has a higher reliability for the next assigned task, i.e., for each compute unit i, $R_i(k_i) > R_i(k_i + x)$, $\forall x \in \mathbb{R} \mid x > 0$. Therefore, the reliability of the VSUs will also be higher, and consequently that yields higher compute units collective reliability.

Furthermore, different compute units collective provisioning strategies also result different non-functional properties of the formed compute units collective. In Table 7.2, we show the average cost and response time of the compute units collectives obtained from the three strategies. Here, the cost is defined as the sum of the execution cost of all members compute units in each compute units collective, while the response time is defined as the duration since the task is assigned to the deployed compute units collective until all members compute units of the compute units collective finish their roles. In these experiments, the *greedy (cost-optimized)* strategy provides 16.70% and 8.35% cheaper

Compute units collective Provisioning Strategies	$avg(cost)$	$avg(response\ times)$
Uniform distribution	7.20	13.585
Fastest response	7.92	11.775
Greedy (cost-optimized)	6.60	12.276

Table 7.2: Compute units collective cost and response times

compute units collectives compared to the *fastest response* strategy and to the *uniform distribution* strategy respectively. For the response time, the compute units collectives provided by the *fastest response* strategy perform the tasks 13.32% and 4.08% faster compared to the compute units collectives provided by the *uniform distribution* strategy and the *greedy (cost-optimized)* strategy respectively.

7.5 Chapter Summary

In this chapter, we presented our framework for analyzing the reliability of compute units collectives for executing tasks in an HCS. We discussed reliability models for individual compute units in the context of task-based executions. Using these models, together with the collective dependency model described in Section 3.3.4, we presented a step-by-step approach for measuring the reliability of a running compute units collective.

Our experiments showed how our reliability analysis technique can be used to obtain insights for improving system's reliability by analyzing different configurations of the HCS. Furthermore, each problem domain has its own requirements with respect to the non-functional properties. Reliability analysis with different compute units collective provisioning strategies help application designers to decide the desirable trade-offs between the reliability and other non-functional properties gained by certain provisioning strategies in a particular problem domain.

CHAPTER 8

Conclusions and Future Work

This last chapter summarizes the results of our work in Section 8.1. Then we revisit again research questions formulated in Section 1.3 and discuss how our work address those issues. Finally, Section 8.3 highlights an outlook for the future research in the context of hybrid human-machine computing.

8.1 Summary

Our work focuses on three important aspects of Hybrid Human-Machine Computing Systems (HCSs): (i) the provisioning of compute units collectives, (ii) the monitoring of a running system, and (iii) the analysis of reliability of task executions by the provisioned compute units collectives. First, in Chapter 3, we presented our architectural view of HCSs and defined models for HCSs that operate based on requested tasks. We developed a platform presented in Chapter 4 based on these models, and prototyped our provisioning, monitoring, and reliability analysis frameworks on this platform.

In Chapter 5, we presented our framework for the quality-aware provisioning of compute units collectives using diverse types and sources of compute units. Our framework contains a provisioning middleware, which controls the provisioning processes, and executes formation engines containing algorithms for quality-aware formation of compute units collectives. We proposed some algorithms for finding optimized formations of compute units collectives considering both, consumer-defined quality requirements, and properties of the discovered compute units. We conducted experiments to study the characteristics of different algorithms, which could be utilized to cater different system needs. In the experiments, we also studied the sensitivity of the formation algorithms with respect to the consumer-defined quality optimization objectives.

We presented, in Chapter 6, our approach for monitoring HCSs, which consist of human-based, software-based, and thing-based subsystems. We tackled challenges dealing with heterogeneous events and metrics emitted by those diverse subsystems.

Moreover, we used Quality of Data (QoD) concept to enable more efficient monitoring of HCSs according to the consumer's requirements. We ran monitoring experiments using monitoring data derived from real world scenarios. Our experiments demonstrated that our monitoring framework is useful to model and measure complex metrics from a running HCS. Furthermore, we showed benefits for both, monitoring clients, and providers, in applying QoD-aware data delivery on HCS monitoring.

In Chapter 7, we presented our approach to analyze the reliability of compute units collectives that consists of human- and machine-based compute units to execute tasks. Our framework is capable to deal with the reliability measurement of compute units collectives, which are dynamically provisioned on-demand using various strategies from large-scale human and machine compute units pools. We first discussed models for measuring the reliability of individual compute units on a task basis. Then we presented the underlying models of compute units collectives. Based on these models we proposed a framework to measure the reliability of compute units collectives. We exemplified our reliability analysis approach in a simulated infrastructure maintenance scenarios. The results of our experiments showed that our framework is beneficial to measure the reliability of the compute units collectives and to obtain insights for improving the collective's quality.

8.2 Research Questions Revisited

In this section, we revisit again our formulated research questions and issues related to the questions. We outline how our main contributions address those issues and what are the limitations.

Research Question 1: *How can we provide a collective of diverse compute units for executing tasks in an HCS considering the consumer-defined requirements?*

A provisioning framework for compute units collectives should consider the consumer-defined quality requirements, which represent functional capabilities requirements as well as non-functional constraints. We modeled the functional and non-functional requirements of a task request in Section 3.3, and developed a strategy, as discussed in Section 5.2, for fulfilling the roles required to execute the task while honoring the requirements. For dealing with non-precise requirements, we employed the concept of fuzzy logic, and optimized the role fulfillment using fuzzy grade membership functions and operations as presented in Section 5.3.1.

Our provisioning framework presented in Section 5.2 allows us to employ different compute units collectives formation algorithms. The formation approach can be used to provision compute units collectives prior to runtime, or to re-provision a running compute units collective due to, e.g., adaptation, as discussed in Section 5.5. For finding (semi-)optimal formation of compute units collectives in a huge search space, we developed heuristics based on greedy and Ant Colony Optimization approaches as presented in Section 5.4.

The heuristic algorithms are controlled by consumer-defined quality optimizing preferences with respect to the functional capability, connectedness, response time, and cost of the compute units. However, the heuristic constructs, e.g., the local fitness and objective value of a solution, can be extended to include more properties.

Research Question 2: *How can an HCS with diverse metrics models and diverse subsystems be effectively monitored?*

An HCS involves diverse types of compute units and different communication technologies, which have to be taken into account by the monitoring system. An approach to deal with such heterogeneity is required to effectively monitor HCSs. We proposed a multi-tier monitoring framework to deal with such heterogeneity, as discussed in Section 6.3.

An HCS consists of different subsystems, each brings along various metrics, which could have corresponding metrics from other subsystems with different semantics. In Section 6.2.1, we introduced different classes of metric measurement to relate corresponding metrics from different subsystems and bring them together as a unified metric to enable system-wide monitoring.

These related metrics from different subsystems of an HCS may have different qualities. Furthermore, different monitoring clients may also require different qualities of monitoring data. In Section 6.2.2, we brought along the concepts of Quality-of-Data and applied them in the context of HCS monitoring.

There are many countless use-cases of a monitoring framework, from design improvement to operation, and post-evaluation of the systems. In our presented framework, we exemplified a rather limited adaptation engine to showcase how monitoring data can be used for improving a running collective. Interested readers may refer to other work, such as [170, 171], for more comprehensive intelligent adaptive systems.

Research Question 3: *How to measure the reliability of an HCS, which consists not only machine-based compute units but also human-based compute units?*

Traditional reliability measurement for machine-based compute units is expressed as a function in a continuous time space. Such approach is not suitable for human-based computing, because most human-based compute units do not operate continuously. In Section 7.2.1, we modeled the reliability of individual human-based compute units on a task basis and apply it to measure the reliability of mixed compute units collectives.

Inter-dependencies between system's elements greatly affect the reliability of the system. We modeled the dependency of a running compute units collective using the collective dependency model described in Section 3.3.4. Such model can be developed from the ground up, or inferred from the process model, e.g., a workflow. We then applied this model and proposed a framework for analyzing the reliability of compute units collectives in Section 7.3.

The provisioning of machine-based and human-based compute units can be made on-demand from a virtually large pool of available compute units. Typically, redundancy

structures can be employed so that when a failure occurred on a running compute unit, another compute unit can be selected to replace. The reliability analysis for cloud-based compute units collectives must take into account this provisioning model. We discussed this reliability structure in Section 7.3.1 using the notion of virtual standby units, and propose a mechanism to analyze the reliability of such structures.

Our proposed reliability analysis approach centers around a task model, hence, yielding the reliability of compute units collectives provided by the system to execute a particular task type. We retained the aggregation analysis to measure the overall system reliability for various task types as a future work.

8.3 Future Work

Our work presented in this thesis is part of our ongoing research in the field of hybrid human-machine computing. While this thesis furnished solutions for important issues, there are still a plentiful of compelling challenges in this domain. Here we list some of interesting issues for future work.

- Harnessing the full computational power of the online collective intelligence into enterprise ecosystems is a challenging task. Many efforts have been done to address this challenge, e.g., [81, 32]. However, still we lack a capability to seamlessly integrate collective intelligence systems into executable business processes. To obtain such process, we would need novel composition notations and interaction protocols, which consider various aspects of collective intelligence, such as collective structures, quality improving techniques, e.g., qualification, redundancy, reviewing, etc., privacy, security, data provenance, and so on.

- Our approaches centers around the structure of compute units collectives modeled using a role-based task model and a collective dependency. This implies that our approaches are suitable for systems with collectives that have known structures. Dealing with unstructured collectives is a challenging task. In such unstructured collectives, issues such as ad-hoc activities, non-deterministic processes, quality requirements modeling, etc., may arise and require further research.

- Reliability is only one, yet important, quality measure for dependable hybrid human-machine computing. Future work includes modeling and measuring various dependability metrics such as availability, performance, performability, and quality of results in the context of HCSs.

- In our monitoring framework, we present metric models allowing the correlation of different metrics, e.g., from machine-based compute units to human-based compute units. Many researches have been conducted to define metrics for human, such as key performance indicators in human resources management, e.g., [172]. However, research on human metrics related to the computational power of human still need further exploration. Several basic definitions of human-based metrics have

been proposed, e.g., [36, 173]. Still, we lack a comprehensive study for metrics of human-based computing.

APPENDIX A

Prototype Documentation

This appendix provides a documentation of our platform *Runtime and Analytics for Hybrid Computing Systems* (RAHYMS). This platform is a prototype of our proposed architecture, models, and frameworks. The platform serves as a proof-of-concept to showcase a realization of quality-aware and reliable Hybrid Human-Machine Computing Systems (HCSs).

In this appendix, first we describe how to get started with the platform. Afterwards, we discuss details of the operation of the platform in the simulation-mode, i.e., how to configure the simulation, and in the interactive-mode, i.e., how to use the Application Programming Interfaces (APIs).

A.1 Getting Started

A.1.1 Overview

This prototype is available as an open-source project, and can be cloned from a GitHub repository `https://github.com/tuwiendsg/RAHYMS`.

The project is developed using Java SDK 1.7, and can be built and deployed using maven. The root project is named *hcu* (stands for *hybrid compute units*). The *hcu* project contains the following sub-projects in their respective directories:

- `hcu-cloud-manager` contains a component to manage compute units including their functional capabilities and non-functional properties, a generator to populate the pool of compute units and to generate task requests in simulation-mode, and a compute unit discovery service. This sub-project also contains tool for calculating the reliability of the individual compute units as well as compute units collectives based on the property of the managed compute units.

- `hcu-common` contains utilities required by the platform, such as the models for compute units and tasks, interfaces to connect different component in the HCS, fuzzy libraries, configuration reader utilities, and tracer utilities.

- `hcu-composer` contains models and library for the formation engine to provision the compute units collectives.

- `hcu-external-lib` contains some adapted external libraries, i.e., GridSim and JSON for Java.

- `hcu-monitor` contains the code for the monitoring framework of HCS. It contains utilities, e.g., to create and deploy monitoring agents, to define, publish, and subscribe metrics, and also a Drools-based rule engine.

- `hcu-rest` contains a Jetty-based Web server for running the interactive-mode. It creates three HTTP services: one service runs the REST API server, one service provides the Web user interface, and another one provides a REST API playground developed using Swagger.

- `hcu-simulation` contains code for running a simulation using GridSim framework.

Additionally, the `smartcom` project is also available in the root as a tool for virtualizing communication with compute units. This project is adopted from the `smartcom` repository available online [1].

A.1.2 Building

For each root project and sub-projects, a maven configuration is provided to allow easy building and importing to an IDE for Java language. Before building the `hcu` project, we first need to build the required `smartcom` project. To build everything, run the following maven commands from the root directory of the repository:

```
1 $ cd smartcom
2 $ mvn install
3 $ cd ../hcu
4 $ mvn install
```

The jar files should now have been created by maven in each projects under the `target` directories. Particularly, two jar files
`hcu/hcu-simulation/target/hcu-simulation-0.0.1-SNAPSHOT.jar`, and
`hcu/hcu-rest/target/hcu-rest-0.0.1-SNAPSHOT.jar`
contain main classes for running the program in simulation- and interactive-mode respectively.

[1] `https://github.com/tuwiendsg/SmartCom`

A.2 Simulation Mode

To run the program in simulation-mode, from the root of the repository simply execute

```
$ java -jar hcu/hcu-simulation/target/hcu-simulation-0.0.1-SNAPSHOT.jar <config-file>
```

where the *<config-file>* argument is the path of the main configuration file.

To execute the program within an IDE, run the main class `at.ac.tuwien.dsg.hcu.simulation.RunSimulation` inside the `hcu-simulation` project with the *<config-file>* as the execution argument.

The main simulation configuration file *<config-file>* is a java properties file containing references to other configuration files specifying a simulation scenario, a composer (i.e., the formation engine) configuration, a tracer configuration, and a monitoring configuration. Listing A.1 shows an example of the main simulation configuration.

```
scenario_config = scenarios/samples/infrastucture-maintenance/scenario.json
composer_config = config/composer.properties
tracer_config   = config/tracer.json
monitor_config  = config/monitor.json # optional
```

Listing A.1: An Example of Main Simulation Configuration

We discuss the content of each configuration as follows.

A.2.1 Scenario Configuration

Our simulation of an HCS consists of two phases:

i) *Initiation Phase* is a phase where compute units are generated with configurable initial properties.

ii) *Execution Phase* is a phase where tasks are generated, and for each task a compute units collective is created to execute the task. The execution phase consists of cycles. In every cycle, the task generator configurations are processed to generate tasks. After a configured number of cycles have passed, the task generation stops, and simulation is finished once all the remaining running tasks are completed.

A simulation scenario mainly has two purposes: it defines how the compute units are generated during the initiation phase, and it defines the generation of task requests during the execution phase. A configuration of a simulation scenario is a json file. An example of a simulation scenario configuration is shown in Listing A.2.

```
{
  "title":"Infrastucture Breakdown Sensing",
  "numberOfCycles":100,
  "waitBetweenCycle":1,    ▶ delay (in simulation time unit) between each cycle
  "service_generator":{
    "basedir":"service-generator/",
    "files":[
      "inspector-generator.json",
      "citizen-generator.json",
```

```
10        "sensor-generator.json"
11      ]
12    },
13    "task_generator":{
14      "basedir":"task-generator/",
15      "files":[
16        "machine-sensing-task-generator.json",
17        "human-sensing-task-generator.json",
18        "mixed-sensing-task-generator.json"
19      ]
20    }
21  }
```

Listing A.2: An Example of Scenario Configuration

Compute Units Generator Configuration

The `service_generator` element in the simulation configuration defines the list configuration for generating compute units together with their provided services (i.e., functional capabilities) and their properties. The `basedir` specifies the directory in which the compute units generator locates the specified `files` list. In Listing A.3, we exemplify a compute units generator configuration annotated to describe the purpose of the configuration. This example shows a generation of citizens as compute units.

```
1  {
2    "seed":1001,    ▶ random number generator seed
3    "numberOfElements":200,    ▶ number of compute units generated
4    "namePrefix":"Citizen",
5    "connection":{
6      "probabilityToConnect":0.4,  ▶ probabiity of a compute unit connected to others
7      "weight":<distribution-config>
8    },
9    "services":[    ▶ the functional services provided by each generated compute unit
10     {
11       "functionality":"DataCollection",
12       "probabilityToHave":0.7,    ▶ probability the compute unit has this
             functionality
13       "properties":[    ▶ functionality-specific properties
14         <property-config>,
15         ...
16       ]
17     },
18     ...
19   ],
20   "commonProperties":[   ▶ non-functional properties
21     <property-config>,
22     ...
23   ]
24 }
```

Listing A.3: An Example of Compute Units Generator Configuration

The *<distribution-config>* defines how a value should be populated with a random number generator, while the *<property-config>* specifies how each property is defined. They are defined in Listing A.4 and Listing A.4 respectively.

```
1 <distribution-config> ::=
```

```
{
  "class":"<distribution-class-name>",
  "params":[...],
  "sampleMethod":"..."
  "mapping":{    ▶ optional
    "0":"<mapped-value-0>",
    "1":"<mapped-value-1>",
    ...
  }
}
```

Listing A.4: Distribution Configuration

The *<distribution-class-name>* is the random number generator class which will be used to generate the random values. It can be any of distribution classes available from Apache Common Math package `org.apache.commons.math3.distribution`[2].

Other distribution classes can also be used by specifying a fully-classified class name. The `params` entry specifies the parameters required the instantiate the distribution class, for example, `NormalDistribution` class can be instantiated using a constructor with three numbers, e.g., $[0.30, 0.10, 1.0E-9]$, which define mean, standard deviation and inverse cumulative distribution accuracy respectively. The `sampleMethod` is a zero-argument method that should be invoked for getting the random values, the default is `sample` for the Apache Common's distribution classes. The optional `mapping` entry defines a mapping from an integer number distribution to a certain value, e.g., a string value.

```
<property-config> ::=
{
  "name":"<property-name>",  ▶ e.g., "location", "cost", etc.
  "probabilityToHave":1.0,  ▶ probability the compute unit has this property
  "type":"<property-type>",  ▶ can be "metric", "skill", or "static"
  "value":<distribution-config>   ▶ required for other than "metric" types
  "interfaceClass":"<property-type>"  ▶ required for "metric" type
}
```

Listing A.5: Property Configuration

A property can be of three types: `metric` property, which defines a property whose value can be retrieved externally, `skill` property, which defines the functional capability of a human-based compute units, and `static` is for all other properties (note that despite of the name, the `static` property value can still be modified by calling the property's setter method during runtime). For `metric` property, the `interfaceClass` entry defines the class implementing `MetricInterface` that provides the value of the property.

Task Generator Configuration

The `task_generator` element in the simulation configuration defines the list configuration for generating tasks at each cycle during the execution phase. The way how

[2] `https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/distribution/package-summary.html`

`basedir` and `files` list work is the same as in the compute units generator configuration. Listing A.6 exemplifies an annotated task generator configuration.

```
{
    "seed": 1001,    ▶ random number generator seed
    "taskTypes": [   ▶ list of task types that should be generated
        {
            "name": "HumanSensingTask",
            "description": "An explanation of the task",
            "tasksOccurance": {*\textit{<distribution-config>}*},  ▶ number of tasks
                generated at each cycle
            "load": {<distribution-config>}, ▶ to simulate how long the task will be
                executed by a unit
            "roles": [  ▶ list of roles for the task
                {
                    "functionality": "DataCollection",  ▶ a functional requirement for
                        the role
                    "probabilityToHave": 1.0,   ▶ probability the role has this
                        functional requirement
                    "relativeLoadRatio": 1.0,   ▶ effective load = relative load *task
                        load
                    "dependsOn": ["...", ...],  ▶ a list of role functionality that
                        this role depends on (collective dependency)
                    "specification": [  ▶ role-level non-functional constraints
                        <specification-config>,
                        ...
                    ]
                },
                ...
            ],
            "specification": [  ▶ task-level non-functional constraints
                <specification-config>,
                ...
            ]
        },
        ...  ▶ multiple task types can be defined
    ]
}
```

Listing A.6: An Example of Task Generator Configuration

The *<distribution-config>* is similar to the one used in the compute units generator configuration. The *<specification-config>* defines non-functional constraints as specified in Listing A.7.

```
<specification-config> ::=
{
  "name":"<property-name>",  ▶ e.g., “location”, “cost”, etc.
  "probabilityToHave":1.0,   ▶ probability the requirement has this constraint
  "type":"<property-type>",  ▶ can be “metric”, “skill”, or “static”
  "value":"<distribution-config>",
  "comparator": "<comparator-class>"
}
```

Listing A.7: A Specification of Non-Functional Constraints

The *<comparator-class>* is a fully-qualified name of a class implementing the `java.util.Comparator` interface. Several comparator classes are provided in `at.ac.tuwien.dsg.hcu.common.sla.comparator` package: `StringComparator`, `NumericAscendingComparator`, `NumericDescendingComparator`, and `FuzzyComparator`.

A.2.2 Formation Engine Configuration

A formation engine configuration is a java properties file specifying the algorithm used by the formation engine, and the parameters required by the algorithms. Listing A.8 shows a snippet example of composer configuration. Currently, available formation algorithms are

- `FairDistribution` algorithm, which distributes tasks uniformly to all qualified compute units,
- `PriorityDistribution` algorithm, which distributes tasks based on the priority of each compute unit specified in `assignment_priority` property, e.g., a compute unit with priority equals to 2 has twice probability to be assigned to tasks compared to compute units with priority equals to 1,
- `EarliestResponse` algorithm, which assigns tasks to compute units with the earliest estimated response time (e.g., the *first come first serve* strategy),
- `GreedyBestFitness` algorithm is a greedy heuristic strategy, which processes each task role iteratively, and for each role a compute unit with the best local fitness value is selected,
- `GreedyHillClimbing` algorithm finds an initial solution similarly as the `Greedy BestFitness` algorithm, and refines the solution further using a *hill climbing* technique, the number of cycles for hill climbing is specified using `maximum_number_of_cycles` parameter,
- `ACOAlgorithm` algorithms, which find the best solution using Ant Colony Optimization. Currently the following variants of ACO algorithms are supported: `AntSystemAlgorithm`, `MinMaxAntSystemAlgorithm`, and `AntColonySystemAlgorithm`. ACO algorithms have many configurable parameters. An example of complete composer configuration including all the parameters can be found in `config/composer.properties` inside the `hcu-composer` project.

```
1 algorithm = ACOAlgorithm
2 aco_variant = AntSystemAlgorithm
3 #aco_variant = MinMaxAntSystemAlgorithm
4 #aco_variant = AntColonySystemAlgorithm
5
6 #algorithm = FairDistribution
7 #algorithm = PriorityDistribution
8 #algorithm = EarliestResponse
9 #algorithm = GreedyBestVisibility
10 #algorithm = GreedyLocalSearch
```

Listing A.8: An Example of Formation Engine Configuration

A.2.3 Tracer Configuration

The tracer configuration is a json file, which specifies the location of trace files (in CSV format) generated during runtime. There are two default tracers, named `reliability` and `composer` tracers, which are used by the formation engine and reliability analysis engine, respectively, for generating the traces of compute units collectives formation created and the reliability measurement for each task execution.

```
[
    {
        "name": "composer",
        "file_prefix": "traces/composer/composer-sample-",
        "class": "at.ac.tuwien.dsg.hcu.composer.ComposerTracer"
    },
    {
        "name": "reliability",
        "file_prefix": "traces/reliability/reliability-sample-",
        "class": "at.ac.tuwien.dsg.hcu.cloud.metric.helper.ReliabilityTracer"
    }
]
```

Listing A.9: An Example of Tracer Configuration

A custom tracer can be created by creating a new class extending `at.ac.tuwien.dsg.hcu.util.Tracer`, and adding a new corresponding entry in the tracer configuration. The new tracer can be invoked anywhere within the program by calling `Tracer.getTracer("<tracer-name>")`.

A.3 Interactive Mode

The following command can be executed to run the program in interactive-mode:

```
$ java -jar hcu/hcu-rest/target/hcu-rest-0.0.1-SNAPSHOT.jar <config-file>
```

where the *<config-file>* argument is the path of the main configuration file.

Within an IDE, the interactive-mode can be started by running the main class `at.ac.tuwien.dsg.hcu.rest.RunRestServer` inside the `hcu-rest` project with the *<config-file>* as the execution argument.

The main configuration file for interactive-mode contains HTTP server configuration, as well as the composer configuration for the formation engine. Note that currently we do not yet support monitoring and reliability analysis in interactive-mode.

Listing A.10 shows an example of configuration for interactive mode. The formation engine configuration defined in `composer_config` has the same format as the `composer_config` in the simulation-mode.

```
SERVER_PORT = 8080
SERVER_HOST = localhost
REST_CONTEXT_PATH = rest
WEBUI_CONTEXT_PATH = web-ui
SWAGGER_CONTEXT_PATH = rest-ui

composer_config = config/composer.properties
```

Listing A.10: An Example of Configuration for Interactive-Mode

Once, the program is started in interactive-mode, the Jetty-based HTTP server is started and listening on the port specified in the configuration. Afterwards, the services can be accessed from

- `http://<SERVER_HOST>:<SERVER_PORT>/<REST_CONTEXT_PATH>` for the RESTful *Application Programming Interfaces* (APIs),
- `http://<SERVER_HOST>:<SERVER_PORT>/<WEBUI_CONTEXT_PATH>` for the *Web User Interface*, and additionally
- `http://<SERVER_HOST>:<SERVER_PORT>/<SWAGGER_CONTEXT_PATH>` for the REST *API playground* based on Swagger.

Note that current prototype implementation of the interactive-mode does not expose full capabilities of the underlying models and framework as found in the simulation-mode. We discuss the APIs provided by our platform as follows.

A.3.1 Application Programming Interface

The Application Programming Interfaces (API) provided by the platform is a RESTful API, which provides CRUD (create, read, update, delete) operations on four entities: `unit`, `task`, `collective`, and `task_rule`.

Below is a list of applicable information for all APIs:

Request URL prefix
`http://<SERVER_HOST>:<SERVER_PORT>/<REST_CONTEXT_PATH>/api`
default: `http://localhost:8080/rest/api`

`POST` and `PUT` parameters encoding (in the request body)
`application/x-www-form-urlencoded`

HTTP response codes
`200`: Successful
`201`: Created successfully
`404`: Error, entity not found
`409`: Error, entity already exists

Response body encoding
`application/json`

The `unit` and `task` entities and their API operations are described as follows. Documentation of API operations for other entities can be viewed online from the Swagger API playground provided by the platform. When an API excepts a URL parameter, it is shown here inside curly brackets. Actual request should not include the brackets in the URL.

Operations on `unit`

a) **`GET /unit`** ▶ *List all units*
Response body on success:

```
[
  {
    "name": "...",
    "email": "...",
    "rest": "...",
    "services": [
      "...", ...
    ],
    ...
  },
  ...
]
```

Note:
- `rest` is the REST service URL for software-based compute units
- `services` is a list of functional capabilities provided by the compute units

b) **`GET /unit/{email}`** ▶ *Find a unit by email*
Response body on success:

```
{
  "name": "...",
  "email": "...",
  "rest": "...",
  "services": [
  "...", ...
  ],
  "elementId": 1
}
```

Note: refer to note for `GET /unit`

c) **`POST /unit`** ▶ *Create a new unit*
Parameters:
- `email` (string)
- `name` (string)
- `rest` (string, optional)
- `services_provided` (string): a comma separated string containing a list of functional capabilities provided by the compute unit, e.g., "`DataCollection, DataAssessment`".

d) **`PUT /unit/{email}`** ▶ *Update an existing unit specified by email*
Parameters:
- `email` (string)
- `name` (string, optional)
- `rest` (string, optional)
- `services_provided` (string): a comma separated string containing a list of functional capabilities provided by the compute unit

Response body on success: refer to response body for `POST /unit`

e) **`DELETE /unit/{email}`** ▶ *Delete an existing unit specified by email*
Response body on success: None

Operations on `task`

In this API, we simplify the task entity model. Each task request has `tag` (e.g., a category) and `severity` (e.g., `'NOTICE'`, `'WARNING'`, `'CRITICAL'`, `'ALERT'`, or `'EMERGENCY'`) properties. When the task request is processed, it is expanded using `task_rule` to a more complete task specification containing the functional capabilities (i.e., services) required to execute the tasks. Here, one service corresponds to one task role. Currently, we do not support updating and deleting a task, because the task is immediately assigned to and executed by the provisioned compute units collectives.

a) **`GET /task`** ▶ *List all tasks*
Response body on success:

```
[
  {
    "id": 1,
    "name": "...",
    "content": "...",
    "severity": "...",
    "tag": "...",
    "timeCreated": "...",
    "collectiveId": 1
  },
  ...
]
```

Note:
- `id` is an auto-generated id of the task
- `collectiveId` is the id of the compute units collective provisioned to execute the task

b) **`GET /task/{id}`** ▶ *Find a task by id*
Response body on success:

```
{
  "id": 1,
  "name": "...",
  "content": "...",
  "severity": "...",
  "tag": "...",
  "timeCreated": "...",
  "collectiveId": 1
}
```

Note: refer to note for `GET /task`

c) **`POST /task`** ▶ *Submit a new task request*
Parameters:
- `name` (string): Task's name
- `content` (string): Task's content description
- `tag` (string): Task's tag, e.g., a category

- `severity` (SeverityLevel) = [`'NOTICE'`, `'WARNING'`, `'CRITICAL'`, `'ALERT'`, or `'EMERGENCY'`]: Task's severity

Bibliography

[1] Luis Von Ahn. Human computation. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 418–419. IEEE, 2009.

[2] Wikipedia. Human-based computation - wikipedia. Website, February 2016. `https://en.wikipedia.org/wiki/Human-based_computation`.

[3] Hyunjung Park and Jennifer Widom. Crowdfill: Collecting structured data from the crowd. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 577–588. ACM, 2014.

[4] Aditya Parameswaran, Stephen Boyd, Hector Garcia-Molina, Ashish Gupta, Neoklis Polyzotis, and Jennifer Widom. Optimal crowd-powered rating and filtering algorithms. *Proceedings Very Large Data Bases (VLDB)*, 2014.

[5] CrowdFlower. Crowdflower content moderation. Website, February 2016. `http://www.crowdflower.com/type-content-moderation`.

[6] Sharoda A Paul, Lichan Hong, and Ed H Chi. What is a question? crowdsourcing tweet categorization. *CHI 2011*, 2011.

[7] Ryan G Gomes, Peter Welinder, Andreas Krause, and Pietro Perona. Crowdclustering. In *Advances in neural information processing systems*, pages 558–566, 2011.

[8] Victor Naroditskiy, Iyad Rahwan, Manuel Cebrian, and Nicholas R Jennings. Verification in referral-based crowdsourcing. *PloS one*, 7(10):e45924, 2012.

[9] Quora. Quora. Website, February 2016. `http://www.quora.com/`.

[10] Yahoo! Yahoo! answers. Website, February 2016. `https://answers.yahoo.com/`.

[11] Seth Cooper, Firas Khatib, Adrien Treuille, Janos Barbero, Jeehyung Lee, Michael Beenen, Andrew Leaver-Fay, David Baker, Zoran Popović, et al. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, 2010.

[12] Justin Wolfers and Eric Zitzewitz. Prediction markets. Technical report, National Bureau of Economic Research, 2004.

[13] Wikipedia. Wikipedia. Website, February 2016. http://www.wikipedia.org/.

[14] Wenjun Wu, Wei-Tek Tsai, and Wei Li. Creative software crowdsourcing: from components and algorithm development to project concept formations. *International Journal of Creative Computing*, 1(1):57–91, 2013.

[15] Anhai Doan, Raghu Ramakrishnan, and Alon Y Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4):86–96, 2011.

[16] Amazon. Amazon mechanical turk. Website, February 2016. http://www.mturk.com/.

[17] Cloudcrowd. Website, 2013. http://www.cloudcrowd.com/.

[18] John G Breslin, Alexandre Passant, and Stefan Decker. Social web applications in enterprise. In *The Social Semantic Web*, pages 251–267. Springer, 2009.

[19] Daniel Schall, Benjamin Satzger, and Harald Psaier. Crowdsourcing tasks to social networks in bpel4people. *World Wide Web*, pages 1–32, 2012.

[20] Djellel Eddine Difallah, Gianluca Demartini, and Philippe Cudré-Mauroux. Pick-a-crowd: tell me what you like, and i'll tell you what to do. In *Proceedings of the 22nd international conference on World Wide Web*, pages 367–374. International World Wide Web Conferences Steering Committee, 2013.

[21] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, Aristides Gionis, and Stefano Leonardi. Online team formation in social networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 839–848. ACM, 2012.

[22] F. Giunchiglia, V. Maltese, S. Anderson, and D. Miorandi. Towards hybrid and diversity-aware collective adaptive systems. 2013.

[23] Karim Benouaret, Raman Valliyur-Ramalingam, and François Charoy. Crowdsc: Building smart cities with large-scale citizen participation. *Internet Computing, IEEE*, 17(6):57–63, 2013.

[24] Thomas W Malone, Robert Laubacher, and Chrysanthos Dellarocas. The collective intelligence genome. *IEEE Engineering Management Review*, 38(3):38, 2010.

[25] U-test, industrial case studies. Website, February 2016. http://www.u-test.eu/use-cases/.

[26] Eric Simmon, Kyoung-Sook Kim, Eswaran Subrahmanian, Ryong Lee, Frederic de Vaulx, Yohei Murakami, Koji Zettsu, and Ram D Sriram. A vision of cyber-physical cloud computing for smart networked systems. *NIST, Aug*, 2013.

[27] Alexander Smirnov, Alexey Kashevnik, and Andrew Ponomarev. Multi-level self-organization in cyber-physical-social systems: Smart home cleaning scenario. *Procedia CIRP*, 30:329–334, 2015.

[28] Enzo Morosini Frazzon, Jens Hartmann, Thomas Makuschewitz, and Bernd Scholz-Reiter. Towards socio-cyber-physical systems in production networks. *Procedia CIRP*, 7:49–54, 2013.

[29] Ashish Agrawal, Mike Amend, Manoj Das, Mark Ford, Chris Keller, Matthias Kloppmann, Dieter König, Frank Leymann, et al. WS-BPEL extension for people (BPEL4People). *V1. 0*, 2007.

[30] D. Jordan et al. Web Services business Process Execution Language (WS-BPEL) 2.0. *OASIS Standard*, 11, 2007.

[31] Gioacchino La Vecchia and Antonio Cisternino. Collaborative workforce, business process crowdsourcing as an alternative of bpo. In *Current Trends in Web Engineering*, pages 425–430. Springer, 2010.

[32] Bikram Sengupta, Anshu Jain, Kamal Bhattacharya, Hong-Linh Truong, and Schahram Dustdar. Collective problem solving using social compute units. *International Journal of Cooperative Information Systems*, 22(04):1341002, 2013.

[33] Hong-Linh Truong and Schahram Dustdar. Context-aware programming for hybrid and diversity-aware collective adaptive systems. In *Business Process Management Workshops*, pages 145–157. Springer, 2014.

[34] Hong-Linh Truong, Hoa Khanh Dam, Aditya Ghose, and Schahram Dustdar. Augmenting complex problem solving with hybrid compute units. In *Service-Oriented Computing–ICSOC 2013 Workshops*, pages 95–110. Springer, 2014.

[35] Panagiotis G Ipeirotis and John J Horton. The need for standardization in crowdsourcing. In *Proceedings of the workshop on crowdsourcing and human computation at CHI*, 2011.

[36] Mohammad Allahbakhsh, Boualem Benatallah, Aleksandar Ignjatovic, Hamid Reza Motahari-Nezhad, Elisa Bertino, and Schahram Dustdar. Quality control in crowdsourcing systems: Issues and directions. *IEEE Internet Computing*, 17(2):76–81, 2013.

[37] David Parmenter. *Key performance indicators: developing, implementing, and using winning KPIs*. John Wiley & Sons, 2015.

[38] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: A survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):33, 2015.

[39] Julien Lesbegueries, Amira Ben Hamida, Nicolas Salatgé, Sarah Zribi, and Jean-Pierre Lorré. Multilevel event-based monitoring framework for the petals enterprise service bus: industry article. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 48–57. ACM, 2012.

[40] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM, 2004.

[41] Shirlei Aparecida De Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. Toward an architecture for monitoring private clouds. *Communications Magazine, IEEE*, 49(12):130–137, 2011.

[42] Monica Scannapieco, Paolo Missier, and Carlo Batini. Data quality at a glance. *Datenbank-Spektrum*, 14:6–14, 2005.

[43] Muhammad ZC Candra, Hong-Linh Truong, and Schahram Dustdar. Provisioning quality-aware social compute units in the cloud. In *Service-Oriented Computing*, pages 313–327. Springer, 2013.

[44] Muhammad ZC Candra, Hong-Linh Truong, and Schahram Dustdar. Analyzing reliability in hybrid compute units. In *Collaboration and Internet Computing, 2015 IEEE 1st International Conference on*. IEEE, 2013.

[45] Muhammad ZC Candra, Hong-Linh Truong, and Schahram Dustdar. Modeling elasticity trade-offs in adaptive mixed systems. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2013 IEEE 22nd International Conference on*, pages 21–26. IEEE, 2013.

[46] Muhammad ZC Candra, Rostyslav Zabolotnyi, Hong-Linh Truong, and Schahram Dustdar. Virtualizing software and human for elastic hybrid services. In *Advanced Web Services*, pages 431–453. Springer, 2014.

[47] Ragunathan Raj Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM, 2010.

[48] Logo design, web design and more. design done differently | 99designs. Website, 2012. `http://www.99designs.com/`.

[49] Crowdflower. Website, February 2016. `http://crowdflower.com/`.

[50] Daniel W Barowy, Charlie Curtsinger, Emery D Berger, and Andrew McGregor. Automan: A platform for integrating human-based and digital computation. *ACM SIGPLAN Notices*, 47(10):639–654, 2012.

[51] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. The jabberwocky programming environment for structured social computing. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 53–64. ACM, 2011.

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

[53] Brian Blake. Crowd services: Human intelligence + web services. *IEEE Internet Computing*, 19(3):4–6, 2015.

[54] D. Schall, H.L. Truong, and S. Dustdar. The human-provided services framework. In *10th IEEE Conference on E-Commerce Technology*, pages 149–156. IEEE, 2008.

[55] S. Dustdar and K. Bhattacharya. The social compute unit. *Internet Computing, IEEE*, 15(3):64–69, 2011.

[56] Schahram Dustdar and Hong-Linh Truong. Virtualizing software and humans for elastic processes in multiple clouds–a service management perspective. *International Journal of Next-Generation Computing (IJNGC)*, 2012.

[57] Salvatore Distefano, Giovanni Merlino, and Antonio Puliafito. Sensing and actuation as a service: A new development for clouds. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pages 272–275. IEEE, 2012.

[58] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

[59] Masahide Nakamura, Shuhei Matsuo, Shinsuke Matsumoto, Hiroyuki Sakamoto, and Hiroshi Igaki. Application framework for efficient development of sensor as a service for home network system. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 576–583. IEEE, 2011.

[60] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010.

[61] Juan Luis Pérez and David Carrera. Performance characterization of the servioticy api: an iot-as-a-service data management platform. In *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pages 62–71. IEEE, 2015.

[62] Benny Mandler, Fabio Antonelli, Robert Kleinfeld, Carlos Pedrinaci, Diego Carrera, Alessio Gugliotta, Daniel Schreckling, Iacopo Carreras, Dave Raggett, Marc Pous, et al. Compose–a journey from the internet of things to the internet of services. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pages 1217–1222. IEEE, 2013.

[63] Amit Sheth, Pramod Anantharam, and Cory Henson. Physical-cyber-social computing: An early 21st century approach. *Intelligent Systems, IEEE*, 28(1):78–82, 2013.

[64] Michael Blackstock, Rodger Lea, and Adrian Friday. Uniting online social networks with places and things. In *Proceedings of the Second International Workshop on Web of Things*, page 5. ACM, 2011.

[65] Soegijardjo Soegijoko. A brief review on existing cyber-physical systems for healthcare applications and their prospective national developments. In *Instrumentation, Communications, Information Technology, and Biomedical Engineering (ICICI-BME), 2013 3rd International Conference on*, pages 2–2. IEEE, 2013.

[66] Siddhartha Kumar Khaitan and James D McCalley. Cyber physical system approach for design of power grids: A survey. In *Power and Energy Society General Meeting (PES), 2013 IEEE*, pages 1–5. IEEE, 2013.

[67] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. Iot gateway: Bridgingwireless sensor networks into internet of things. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 347–352. IEEE, 2010.

[68] Project brillo. Website, February 2016. `https://developers.google.com/brillo/`.

[69] Windows iot. Website, February 2016. `https://dev.windows.com/en-us/iot`.

[70] Stefan Nastic, Sanjin Sehic, Duc-Hung Le, Hong-Linh Truong, and Schahram Dustdar. Provisioning software-defined iot cloud systems. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 288–295. IEEE, 2014.

[71] Zhong Liu, Dong-sheng Yang, Ding Wen, Wei-ming Zhang, and Wenji Mao. Cyber-physical-social systems for command and control. *IEEE Intelligent Systems*, (4):92–96, 2011.

[72] Thomas W Malone and Michael S Bernstein. *Handbook of Collective Intelligence*. 2015.

[73] Threadless graphic t-shirt designs: cool funny t-shirts weekly! tees designed by the community. Website, February 2016. `http://www.threadless.com/`.

[74] Home | innocentive. Website, February 2016. `http://www.innocentive.com/`.

[75] Topcoder, inc. | home of the world's largest development community. Website, February 2016. `http://www.topcoder.com`.

[76] Anand P Kulkarni, Matthew Can, and Bjoern Hartmann. Turkomatic: automatic recursive task and workflow design for mechanical turk. In *CHI'11 Extended Abstracts on Human Factors in Computing Systems*, pages 2053–2058. ACM, 2011.

[77] D.C. Brabham. Crowdsourcing as a model for problem solving. *Convergence: The International Journal of Research into New Media Technologies*, 14(1):75, 2008.

[78] M. Vukovic and C. Bartolini. Towards a research agenda for enterprise crowdsourcing. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 425–434, 2010.

[79] Osamuyimen Stewart, Juan M Huerta, and Melissa Sader. Designing crowdsourcing community for the enterprise. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 50–53. ACM, 2009.

[80] Crowdengineering - crowdsourcing customer service. Website, 2012. `http://www.crowdengineering.com/`.

[81] Maja Vukovic, Mariana Lopez, and Jim Laredo. Peoplecloud for the globally integrated enterprise. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 109–114. Springer, 2010.

[82] Charles Petrie. Plenty of room outside the firm [peering]. *Internet Computing, IEEE*, 14(1):92–96, 2010.

[83] DBpedia. Dbpedia. Website, April 2016. `http://wiki.dbpedia.org/`.

[84] Amit Sheth. Citizen sensing, social signals, and enriching human experience. *IEEE Internet Computing*, (4):87–92, 2009.

[85] Bin Guo, Zhu Wang, Zhiwen Yu, Yu Wang, Neil Y Yen, Runhe Huang, and Xingshe Zhou. Mobile crowd sensing and computing: The review of an emerging human-powered sensing paradigm. *ACM Computing Surveys (CSUR)*, 48(1):7, 2015.

[86] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *Proceedings of the 19th international conference on World wide web*, pages 851–860. ACM, 2010.

[87] Jiong Jin, Jayavardhana Gubbi, Slaven Marusic, and Marimuthu Palaniswami. An information framework for creating a smart city through internet of things. *Internet of Things Journal, IEEE*, 1(2):112–121, 2014.

[88] Marlon Dumas, Wil M Van der Aalst, and Arthur H Ter Hofstede. *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, 2005.

[89] A. Agrawal et al. Web Services Human Task (WS-HumanTask), version 1.0. 2007.

[90] Object Management Group (OMG). Business process model and notation 2.0. 2011.

[91] Martin Treiber, Daniel Schall, Schahram Dustdar, and Christian Scherling. Tweetflows: flexible workflows with twitter. In *Proceedings of the 3rd international workshop on Principles of engineering service-oriented systems*, pages 1–7. ACM, 2011.

[92] Hsiao-Hsien Chiu and Ming-Shi Wang. A study of iot-aware business process modeling. *International Journal of Modeling and Optimization*, 3(3):238, 2013.

[93] Stefano Tranquillini, Patrik Spieß, Florian Daniel, Stamatis Karnouskos, Fabio Casati, Nina Oertel, Luca Mottola, Felix Jonathan Oppermann, Gian Pietro Picco, Kay Römer, et al. Process-based design and integration of wireless sensor network applications. In *Business Process Management*, pages 134–149. Springer, 2012.

[94] Sonja Meyer, Andreas Ruppen, and Carsten Magerkurth. Internet of things-aware process modeling: integrating iot devices as business process resources. In *Advanced Information Systems Engineering*, pages 84–98. Springer, 2013.

[95] Alexandru Caracaş and Alexander Bernauer. Compiling business process models for sensor networks. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pages 1–8. IEEE, 2011.

[96] Patrik Spiess, H Vogt, and H Jutting. Integrating sensor networks with business processes. In *Real-World Sensor Networks Workshop at ACM MobiSys*, 2006.

[97] Stephan Haller and Carsten Magerkurth. The real-time enterprise: Iot-enabled business processes. In *IETF IAB Workshop on Interconnecting Smart Objects with the Internet*. Citeseer, 2011.

[98] Merriam-Webster. Definition of provision by merriam-webster. http://www.merriam-webster.com/dictionary/provision.

[99] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, RaüL Sirvent, Jordi Guitart, Rosa M Badia, Karim Djemame, et al. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012.

[100] Rodrigo N Calheiros, Rajiv Ranjan, and Rajkumar Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 295–304. IEEE, 2011.

[101] Sankaran Sivathanu, Ling Liu, Mei Yiduo, and Xing Pu. Storage management in virtualized cloud environment. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 204–211. IEEE, 2010.

[102] Fabio Casati. Promises and failures of research in dynamic service composition. In *Seminal Contributions to Information Systems Engineering*, pages 235–239. Springer, 2013.

[103] Michael Vogler, Johannes Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. Leonore–large-scale provisioning of resource-constrained iot deployments. In *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*, pages 78–87. IEEE, 2015.

[104] Stuart Clayman and Alex Galis. Inox: A managed service platform for interconnected smart objects. In *Proceedings of the workshop on Internet of Things and Service Platforms*, page 2. ACM, 2011.

[105] Gilbert Cassar, Payam Barnaghi, Wei Wang, and Klaus Moessner. A hybrid semantic matchmaker for iot services. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 210–216. IEEE, 2012.

[106] Jong Myoung Ko, Chang Ouk Kim, and Ick-Hyun Kwon. Quality-of-service oriented web service composition algorithm and planning architecture. *Journal of Systems and Software*, 81(11):2079–2090, 2008.

[107] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005.

[108] Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf Steinmetz. Heuristics for qos-aware web service composition. In *Web Services, 2006. ICWS'06. International Conference on*, pages 72–82. IEEE, 2006.

[109] Safina Showkat Ara, Zia Ush Shamszaman, and Ilyoung Chong. Web-of-objects based user-centric semantic service composition methodology in the internet of things. *International Journal of Distributed Sensor Networks*, 2014, 2014.

[110] Thiago Teixeira, Sara Hachem, Valérie Issarny, and Nikolaos Georgantas. Service oriented middleware for the internet of things: a perspective. In *Towards a Service-Based Internet*, pages 220–229. Springer, 2011.

[111] Adil Baykasoglu, Turkay Dereli, and Sena Das. Project team selection using fuzzy optimization approach. *Cybernetics and Systems: An International Journal*, 38(2):155–185, 2007.

[112] D Strnad and N Guid. A fuzzy-genetic decision support system for project team formation. *Applied Soft Computing*, 10(4):1178–1187, 2010.

[113] Syama Sundar Rangapuram, Thomas Bühler, and Matthias Hein. Towards realistic team formation in social networks based on densest subgraphs. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1077–1088. International World Wide Web Conferences Steering Committee, 2013.

[114] Mehdi Kargar, Aijun An, and Morteza Zihayat. Efficient bi-objective team formation in social networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 483–498. Springer, 2012.

[115] Michelle Cheatham and Kevin Cleereman. Application of social network analysis to collaborative team formation. In *Proceedings of the International Symposium on*

Collaborative Technologies and Systems, pages 306–311. IEEE Computer Society, 2006.

[116] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 467–476. ACM, 2009.

[117] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[118] Nagios. Nagios - the industry standard in it infrastructure monitoring. Website, February 2016. `http://www.nagios.org/`.

[119] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[120] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Mela: elasticity analytics for cloud services. *International Journal of Big Data Intelligence*, 2(1):45–62, 2015.

[121] Marcio Barbosa de Carvalho and Lisandro Zambenedetti Granville. Incorporating virtualization awareness in service monitoring systems. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 297–304. IEEE, 2011.

[122] IBM. Monitoring and administering human tasks with websphere business monitor. Website, April 2009. `http://www.ibm.com/developerworks/websphere/library/techarticles/0904_xing/0904_xing.html`.

[123] Felix Freiling, Irene Eusgeld, and Ralf Reussner. Dependability metrics. *Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany*, 2008.

[124] Íñigo Goiri, Ferran Julià, J Oriol Fitó, Mario Macías, and Jordi Guitart. Resource-level qos metric for cpu-based guarantees in cloud providers. In *Economics of Grids, Clouds, Systems, and Services*, pages 34–47. Springer, 2010.

[125] Karthik Lakshmanan, Dionisio De Niz, Ragunathan Rajkumar, and Gines Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 169–178. IEEE, 2010.

[126] Carlo Batini, Cinzia Cappiello, Chiara Francalanci, and Andrea Maurino. Methodologies for data quality assessment and improvement. *ACM Computing Surveys (CSUR)*, 41(3):16, 2009.

[127] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[128] Chrysostomos Zeginis, Konstantina Konsolaki, Kyriakos Kritikos, and Dimitris Plexousakis. Ecmaf: an event-based cross-layer service monitoring and adaptation framework. In *Service-Oriented Computing-ICSOC 2011 Workshops*, pages 147–161. Springer, 2012.

[129] L.R. Varshney, A. Vempaty, and P.K. Varshney. Assuring privacy and reliability in crowdsourcing with coding. In *ITA Workshop*, pages 1–6. IEEE, 2014.

[130] Roi Blanco, Harry Halpin, Daniel M Herzig, Peter Mika, Jeffrey Pound, Henry S Thompson, and Thanh Tran Duc. Repeatable and reliable search system evaluation using crowdsourcing. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 923–932. ACM, 2011.

[131] Shih-Wen Huang and Wai-Tat Fu. Enhancing reliability using peer consistency evaluation in human computation. In *CSCW '13*, pages 639–648. ACM, 2013.

[132] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics*, 1(3):281–308, 2004.

[133] Paolo Bocciarelli, Andrea D'Ambrogio, Andrea Giglio, and Emiliano Paglia. Simulation-based performance and reliability analysis of business processes. In *Proceedings of the 2014 Winter Simulation Conference*, pages 3012–3023. IEEE Press, 2014.

[134] JC Williams. Heart–a proposed method for assessing and reducing human error. In *9th Advances in Reliability Technology Symposium, University of Bradford*, 1986.

[135] E. Hollnagel. *Cognitive reliability and error analysis method (CREAM)*. Elsevier Science, 1998.

[136] William J Kolarik, Jeffrey C Woldstad, Susan Lu, and Huitian Lu. Human performance reliability: on-line assessment using fuzzy logic. *IIE transactions*, 36(5):457–467, 2004.

[137] R Rukšėnas, Jonathan Back, Paul Curzon, and Ann Blandford. Formal modelling of salience and cognitive load. *Electronic Notes in Theoretical Computer Science*, 208:57–75, 2008.

[138] Y.S. Dai, M. Xie, and X. Wang. A heuristic algorithm for reliability modeling and analysis of grid systems. *Systems, Man and Cybernetics*, 37(2):189–200, 2007.

[139] S. Guo, H.Z. Huang, Z. Wang, and M. Xie. Grid service reliability modeling and optimal task scheduling considering fault recovery. *Reliability*, 60(1):263–274, 2011.

[140] Thanadech Thanakornworakij, Raja F Nassar, Chokchai Leangsuksun, and Mihaela Păun. A reliability model for cloud computing for high performance computing applications. In *Euro-Par 2012: Parallel Processing Workshops*, pages 474–483. Springer, 2012.

[141] N. Yadav, V.B. Singh, and M. Kumari. Generalized reliability model for cloud computing. *International Journal of Computer Applications*, 88(14):13–16, 2014.

[142] M. Maybury, R. D'Amore, and D. House. Expert finding for collaborative virtual environments. *Communications of the ACM*, 44(12):55–56, 2001.

[143] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005.

[144] Bikram Sengupta, Anshu Jain, Kamal Bhattacharya, Hong-Linh Truong, and Schahram Dustdar. Who do you call? problem resolution through social compute units. In *Service-Oriented Computing*, pages 48–62. Springer, 2012.

[145] Li-jie Jin, Fabio Casati, Mehmet Sayal, and Ming-Chien Shan. Load balancing in distributed workflow management system. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 522–530. ACM, 2001.

[146] Lotfi Asker Zadeh. The concept of a linguistic variable and its application to approximate reasoning - I. *Information sciences*, 8(3):199–249, 1975.

[147] Richard E Bellman and Lotfi Asker Zadeh. Decision-making in a fuzzy environment. *Management science*, 17(4):B–141, 1970.

[148] L.R. Varshney. Privacy and reliability in crowdsourcing service delivery. In *SRII Global Conference (SRII), 2012 Annual*, pages 55–60. IEEE, 2012.

[149] Frank Spillers and Daniel Loewus-Deitch. Temporal attributes of shared artifacts in collaborative task environments. 2003.

[150] Philipp Zeppezauer, Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Virtualizing communication for hybrid and diversity-aware collective adaptive systems. In *10th International Workshop on Engineering Service-Oriented Applications (WESOA '14), 12th International Conference on Service Oriented Computing, Paris, France*, 2014.

[151] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[152] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[153] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.

[154] Thomas Stutzle and Holger HHH Hoos. Max-min ant system. *Future generations computer systems*, 16(8):889–914, 2000.

[155] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, Apr 1997.

[156] Adrian Mos, Carlos Pedrinaci, Guillermo Alvaro Rey, Jose Manuel Gomez, Dong Liu, Guillaume Vaudaux-Ruth, and Samuel Quaireau. Multi-level monitoring and analysis of web-scale service based applications. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 269–282. Springer, 2010.

[157] S. Frischbier, E. Turan, M. Gesmann, A. Margara, D. Eyers, P. Eugster, P. Pietzuch, and A. Buchmann. Effective runtime monitoring of distributed event-based enterprise systems with asia. In *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on*, pages 41–48. IEEE, 2014.

[158] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011.

[159] Ali Benssam, Jean Berger, Abdeslem Boukhtouta, Mourad Debbabi, Sujoy Ray, and Abderrazak Sahi. What middleware for network centric operations? *Knowledge-Based Systems*, 20(3):255–265, 2007.

[160] J Kephart, J Kephart, D Chess, Craig Boutilier, Rajarshi Das, Jeffrey O Kephart, and William E Walsh. An architectural blueprint for autonomic computing. *IBM White paper*, 2003.

[161] Georgiana Copil, Demetris Trihinas, Hong-Linh Truong, Daniel Moldovan, George Pallis, Schahram Dustdar, and Marios Dikaiakos. Advise–a framework for evaluating cloud service elasticity behavior. In *Service-Oriented Computing*, pages 275–290. Springer Berlin Heidelberg, 2014.

[162] Irene Eusgeld, Felix Freiling, and Ralf H Reussner. *Dependability Metrics*, volume 4909. Springer, 2008.

[163] P.G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD workshop on human computation*, pages 64–67. ACM, 2010.

[164] Zibin Zheng and Michael R Lyu. Collaborative reliability prediction of service-oriented systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 35–44. ACM, 2010.

[165] I. Eusgeld, B. Fechner, F. Salfner, M. Walter, and P. Limbourg. Hardware reliability. *Dependability metrics*, pages 59–103, 2008.

[166] I. Eusgeld, F. Fraikin, M. Rohr, F. Salfner, and U. Wappler. Software reliability. *Dependability metrics*, pages 104–125, 2008.

[167] I. Koren and C.M. Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[168] Benjamin Satzger, Harald Psaier, Daniel Schall, and Schahram Dustdar. Stimulating skill evolution in market-based crowdsourcing. In *Business Process Management*, pages 66–82. Springer, 2011.

[169] X. Zang, H. Sun, and K.S. Trivedi. A bdd-based algorithm for reliability analysis of phased-mission systems. *Reliability, IEEE Transactions on*, 48(1):50–60, 1999.

[170] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, and J.M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009.

[171] Debanjan Ghosh, Raj Sharman, H Raghav Rao, and Shambhu Upadhyaya. Self-healing systems-survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[172] Iveta Gabcanova. Human resources key performance indicators. *Journal of Competitiveness*, 4(1), 2012.

[173] Mirela Riveni, Hong-Linh Truong, and Schahram Dustdar. On the elasticity of social compute units. In *Advanced Information Systems Engineering*, pages 364–378. Springer, 2014.

Glossary

compute unit is a resource providing services capable of processing input data into a more useful information in a (semi-)automated manner.

human-based compute unit is a human actor acting as a compute unit.

machine-based compute unit is a non-human compute unit, i.e., a software-based compute unit or a thing-based compute unit.

software-based compute unit is a compute unit providing software-based services, including software applications, and (virtual-)machines.

thing-based compute unit is a compute unit interacting directly with physical entities, i.e., sensors, actuators, and their gateways.

compute units collective is a construct for a flexible group of human-based and/or machine-based compute units, which can be composed, deployed, and dismissed on-demand for executing tasks.

Hybrid Human-Machine Computing System (HCS) is a system employing humans and machines as compute units, where tasks are distributed to humans and machines, and solutions from both humans and machines are collected, interpreted and integrated.

task is an abstraction of a set of activities and their requirements to be executed by a compute units collective.

activity is an actual piece of work need to be undertaken by a compute unit within the context of a task. A task contains a set of activities, i.e., an activity can be considered as a sub-task.

role is an association of a compute unit and the activity that he/she/it undertakes within a task.