Predictable Execution Of Scientific Workflows Using Advance Resource Reservations

Inauguraldissertation

zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Christoph Langguth
aus Jena, Deutschland
Basel, 2014

Originaldokument gespeichert auf dem Dokumentenserver der Universität Basel
edoc.unibas.ch

Dieses Werk ist unter dem Vertrag “Creative Commons Namensnennung-Keine kommerzielle Nutzung-Keine Bearbeitung 3.0 Schweiz” lizenziert. Die vollständige Lizenz kann unter
creativecommons.org/licenses/by-nc-nd/3.0/ch/
eingesehen werden.

Attribution-NonCommercial-NoDerivs 3.0 Switzerland
(CC BY-NC-ND 3.0 CH)

You are free to:

Share – copy and redistribute the material in any medium or format

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial – You may not use the material for commercial purposes.

NoDerivatives – If you remix, transform, or build upon the material, you may not distribute the modified material.

No additional restrictions – You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Source: http://creativecommons.org/licenses/by-nc-nd/3.0/ch/deed.en Date: 25 February 2014

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät

auf Antrag von

Prof. Dr. Heiko Schuldt, Dissertationsleiter
Prof. Dr. Walter Binder, Korreferent

Basel, den 18.02.2014

Prof. Dr. Jörg Schibler, Dekan

Zusammenfassung

Workflows im wissenschaftlichen Umfeld sind langlaufende und datenintensive Prozesse, welche Operationen von mehreren, geografisch verteilten, Service Providern beinhalten können. Die traditionelle Ausführungsmethode für solche Workflows ist die Verwendung einer einzelnen Workflow Engine, die die gesamte Durchführung einer Prozessinstanz koordiniert und überwacht, wobei aber der aktuelle Zustand der Infrastruktur (z.B. die Auslastung der Rechner oder des Netzwerks) weitestgehend unbekannt oder unberücksichtigt bleibt. Solch zentralisierte Ausführungen können daher zu einer ineffizienten Ressourcennutzung führen – etwa weil große Datenmengen wiederholt über langsame Netzwerkverbindungen gesendet werden – und können keine Garantien für die Quality of Service (QoS) abgeben. Insbesondere kann beispielsweise die parallele Ausführung mehrerer unabhängiger Prozesse zur Überlastung einiger Ressourcen führen, welche die Leistung bzw. den Durchsatz all dieser Prozesse beeinträchtigt.

Unser Ansatz zur Ermöglichung eines vorhersagbaren Verhaltens besteht darin, Ressourcen proaktiv zu verwalten (also vor der Nutzung zu reservieren) und Ausführungen auf mehreren verteilten Workflow Engines zu koordinieren. Dies erlaubt es, die existierenden Ressourcen effizient zu nutzen (beispielsweise indem der bestgeeignete Anbieter einer Operation verwendet wird und bei großen Datentransfers Netzwerklokalität berücksichtigt wird), ohne sie zu überlasten. Gleichzeitig ermöglicht diese Vorgehensweise Vorhersagbarkeit – betreffend Ressourcennutzung, Ausführungsdauer und Kosten – die sowohl Diensteanbietern als auch ihren Nutzern zugute kommt.

Die Beiträge dieser Dissertation sind im Folgenden aufgeführt. Zunächst wird ein formales Modell vorgestellt, bestehend aus Konzepten und Operationen zur Darstellung eines Systems, in welchem Diensteanbieter die zur Ausführung der angebotenen Operationen benötigten Ressourcen kennen und berücksichtigen und in welchem (geplante) Workflow-Ausführungen an den Zustand der Infrastruktur angepasst werden.

Zweitens wird die prototypische Implementierung eines solchen Systems dargelegt, wobei jede Prozessausführung zwei wesentliche Phasen umfasst. In der Planungsphase müssen die Ressourcen für die zukünftige Ausführung bestimmt werden, was durch einen genetischen Algorithmus geschieht. Wir beleuchten dabei konzeptionelle sowie Implementierungsdetails zur Gestaltung der Chromosomen und der Fitness-Funktionen, die benötigt werden, um Ausführungen nach nutzerdefinierten Optimierungskriterien zu planen. In der Ausführungsphase muss das System sicherstellen, dass die tatsächliche Ressourcennutzung mit den erfolgten Reservierungen übereinstimmt. In diesem Kontext wird aufgezeigt, wie eine solche Durchsetzung für verschiedene Arten von Ressourcen erfolgen kann.

Drittens beschreiben wir die Zusammenarbeit dieser Komponenten und das gesamte prototypische System, welches eine auf WSDL/SOAP Web Services, UDDI Registries und Glassfish Application Servern basierende Infrastruktur bildet. Abschließend präsentieren und diskutieren wir die Resultate verschiedener Evaluierungen, die Planung und Ausführung betreffen.

Abstract

Scientific Workflows are long-running and data intensive, and may encompass operations provided by multiple physically distributed service providers. The traditional approach to execute such workflows is to employ a single workflow engine which orchestrates the entire execution of a workflow instance, while being mostly agnostic about the state of the infrastructure it operates in (e.g., host or network load). Therefore, such centralized best-effort execution may use resources inefficiently – for instance, repeatedly shipping large data volumes over slow network connections – and cannot provide Quality of Service (QoS) guarantees. In particular, independent parallel executions might cause an overload of some resources, resulting in a performance degradation affecting all involved parties.

In order to provide predictable behavior, we propose an approach where resources are managed proactively (i.e., reserved before being used), and where workflow execution is handled by multiple distributed and cooperating workflow engines. This allows to efficiently use the existing resources (for instance, using the most suitable provider for operations, and considering network locality for large data transfers) without overloading them, while at the same time providing predictability – in terms of resource usage, execution timing, and cost – for both service providers and customers.

The contributions of this thesis are as follows. First, we present a system model which defines the concepts and operations required to formally represent a system where service providers are aware of the resource requirements of the operations they make available, and where (planned) workflow executions are adapted to the state of the infrastructure.

Second, we describe our prototypical implementation of such a system, where a workflow execution comprises two main phases. In the planning phase, the resources to reserve for an upcoming workflow execution must be determined; this is realized using a Genetic Algorithm. We present conceptual and implementation details of the chromosome layout, and the fitness functions employed to plan executions according to one or more user-defined optimization goals. During the execution phase, the system must ensure that the actual resource usages abide to the reservations made. We present details on how such enforcement can be performed for various resource types.

Third, we describe how these parts work together, and how the entire prototype system is deployed on an infrastructure based on WSDL/SOAP Web Services, UDDI Registries, and Glassfish Application Servers. Finally, we discuss the results of various evaluations, encompassing both the planning and runtime enforcement.

Contents

Zusammenfassung i

Abstract iii

1 Introduction 1

2 System Model 3
2.1 DWARFS at a glance . . . 3
2.1.1 Scientific Workflows . . . 3
2.1.2 Resources, Hosts, Operation Providers . . . 4
2.1.3 Resources and Reservations . . . 5
2.2 Operations and Workflows . . . 6
2.2.1 Data Types . . . 6
2.2.2 Operation . . . 6
2.2.3 Workflows . . . 7
2.3 Resources . . . 9
2.3.1 Resource classes, types, and instances . . . 10
2.3.2 Common resource types . . . 12
2.3.3 Other Resource Types . . . 12
2.4 Resource Usage . . . 13
2.4.1 Usage Blocks . . . 13
2.4.2 Allocations . . . 14
2.4.3 Co-allocations: interdependent allocations . . . 19
2.4.4 Allocation And Co-Allocation Cost . . . 20
2.4.5 Co-Allocation Constraints . . . 21
2.5 Reservations and Infrastructure State . . . 29
2.5.1 Reservations . . . 29
2.5.2 State and State Changes . . . 30
2.6 Workflow Execution in DWARFS . . . 32
2.6.1 Workflow Orchestration and Physical Data Flow . . . 32
2.6.2 Predicting Workflow Executions . . . 34
2.6.3 Data Characteristics . . . 37
2.6.4 Endpoints, Operation Instances, and Workflow Engines . . . 38
2.6.5 Data Transfers . . . 40
2.6.6 Operation Invocation . . . 40
2.7 Workflow Schedules . . . 42
2.7.1 Determining Workflow Schedules . . . 43
2.7.2 Functions relating to Workflow Schedules . . . 45
2.7.3 Workflow Schedule Validity and Reservation . . . 49
2.7.4 Workflow Schedules as Operation Instances . . . 50
2.8 Summary and Discussion . . . 50

3 Planning Under Quality of Service Criteria 55
3.1 QoS Metrics and Goals . . . 55
3.2 Optimization approach . . . 59
3.2.1 Genetic Algorithms in a Nutshell . . . 59
3.3 Chromosome Representation . . . 62
3.3.1 Genes . . . 62
3.3.2 Correspondence with the Formal Model . . . 65
3.3.3 Interpretation of the Chromosome . . . 67
3.3.4 Chromosome as a Variable Dependency Graph . . . 69
3.3.5 Mutations . . . 72
3.3.6 Determining Compliant Co-Allocations . . . 73
3.4 Fitness Functions in DWARFS . . . 79
3.4.1 Semantics and Comparability of Individual Schedule Metrics . . . 80
3.4.2 Multi-objective Optimization . . . 80
3.4.3 Normalized Fitness Functions . . . 81
3.4.4 Bounds . . . 86
3.4.5 Fitness Evaluation . . . 86
3.5 Partitioning and Data Transfers . . . 87
3.5.1 Partitioning . . . 88
3.5.2 Data Transfers . . . 88

4 Reservation Enforcement 95
4.1 Environment and Assumptions . . . 96
4.2 Enforcement of Persistent Resources . . . 98
4.3 Enforcement of Transitory Resources . . . 99
4.4 Bandwidth Enforcement . . . 99
4.5 CPU Enforcement . . . 100
4.5.1 Gathering CPU statistics . . . 101
4.5.2 Influencing CPU shares . . . 102
4.5.3 The Control Loop . . . 103
4.5.4 Fuzzy Logic and Fuzzy Controllers in a Nutshell . . . 104
4.5.5 Implementation Details . . . 109

5 Infrastructure Implementation 115
5.1 Infrastructure . . . 115
5.1.1 Terminology . . . 115
5.1.2 Components Overview . . . 117
5.2 Interactions . . . 119
5.2.1 Service Startup and Registration . . . 120
5.2.2 Workflow Planning . . . 121
5.3 Practical Considerations . . . 122
5.3.1 Ramifications of using a UDDI Registry . . . 122
5.3.2 Glassfish Configuration and Adaptation . . . 123
5.3.3 Requirements for Operational Service Implementations . . . 124
5.3.4 Concurrent Planning and Reservations . . . 125

6 Evaluation **127**
6.1 Planner Evaluation 127
6.1.1 Simulated Deployment 128
6.1.2 Scenario 1: Weather Forecast Workflow 129
6.1.3 Scenario 2: Data Transfer Strategies 142
6.1.4 Discussion 147
6.2 Enforcement Evaluation 148
6.2.1 Scenario 1: Single Process on Amazon Web Services Infrastructure 148
6.2.2 Scenario 2: Multiple Processes on Local Infrastructure 151
6.2.3 Discussion 155

7 Related Work **157**
7.1 Distributed and Scientific Workflow Systems 157
7.2 Planning and Advance Reservations 160
7.3 Monitoring and Prediction 162

8 Conclusion and Outlook **163**
8.1 Summary 163
8.2 Directions for Future Work 164

Bibliography **165**

Index **171**

List of Figures

1.1 Weather Forecast Workflow 1

2.1 Sample workflow (abstract) 3
2.2 Sample workflow, with logical data flow 4
2.3 Screenshots: Resource Monitoring 10
2.4 Example of Resource Classes 10
2.5 Allocation Addition Example 18
2.6 Sample Co-allocation for an operation invocation 19
2.7 Alternative Allocations for a Time-Determining Allocation Constraint . . 23
2.8 Sample Workflow, with Physical Data Flow 33
2.9 Sample Workflow – Timing 34
2.10 Workflow Definitions: Sorting Lists of Numbers 35
2.11 Workflow Scheduling: Operation Instances 44
2.12 Workflow Scheduling: Workflow Engines and Inter-Engine Constraints . . 45
2.13 DWARFS predictability: Metadata Quality and Scheduling Results 53

3.1 Genetic Algorithm: Sample Population and Fitnesses 60
3.2 Genetic Algorithm: Mutation and Crossover 61
3.3 Sample Workflow Chromosome Layout 62
3.4 Correspondence of Workflow Description and Chromosome layout (Simple Workflow) 66
3.5 Correspondence of Workflow Description and Chromosome layout (Complex Workflow Fragment) 66
3.6 Chromosome Interpretation and Interdependencies 67
3.7 Variable Dependency Graph of a simple Workflow 71
3.8 Annotated Variable Dependency Graph Fragment 72
3.9 Simplified Variable Dependency Graph Fragment with Parallelism 74
3.10 Strategies for dependent Co-allocation Variables during Mutations 76
3.11 Graphical and internal Representation of an Allocation 76
3.12 Planning Goals: Duration vs. Termination in Loaded Infrastructure 81
3.13 Normalized Fitness Function 82
3.14 Comparison of various Fitness Functions 85
3.15 Example of Process Fragmentation and Data Transfers 87
3.16 Resource Usage for Inter-Engine Data Transfers 89
3.17 Data Transfer Strategy Alternatives 90
3.18 Extended Data Transfer Genes Layout 91

4.1 Runtime Environment Architectural Overview 96
4.2 CPU shares and overhead 101
4.3 Mapping of Java thread priorities to effective CPU shares on different OS's 102
4.4 Fuzzy Term Definitions 105

4.5 Fuzzy rules evaluation: overspent CPU with dropping tendency 107
4.6 Fuzzy rules evaluation: overspent CPU with rising tendency 108
4.7 System state evolution during CPU share controller run 111
4.8 System state evolution (aggregated shares) 112

5.1 Infrastructure Overview 116
5.2 Components deployed on DWARFS Servers 118
5.3 Infrastructure Interactions: Registration 120
5.4 Infrastructure Interactions: Planning 121
5.5 Overview of WSDL to UDDI mapping, according to [B$^+$01] 123

6.1 Planning Scenario 1: Process Definition and Characteristics 129
6.2 Planning Scenario 1, 50 Processes: Planned Runtimes 132
6.3 Planning Scenario 1: Resource Usages of ireland13 and ireland12 137
6.4 Planning Scenario 1: Evolution of the Planning of Processes 12 and 13 . . . 138
6.5 Planning Scenario 1, 500 Processes: Planned Runtimes 139
6.6 Planning Scenario 1, 500 Processes: Site Network Usages 140
6.7 Planning Scenario 1, 500 Processes: Select Host CPU Usages 141
6.8 Planning Scenario 2: Process Definition and Characteristics 143
6.9 Enforcement: Accounting for Buffering Behavior 151

List of Tables

3.1 Chromosome Interpretation and Relation to Model 68
3.2 Sample Results for Various Fitness Determination Strategies 84

4.1 Evaluation results 112

6.1 Sites Connectivity and Transfer Costs 128
6.2 Planning Scenario 1: Infrastructure Deployment 130
6.3 Planning Scenario 1: Infrastructure Deployment (contd.) 131
6.4 Allocations for Process Fragment (Processes 1 - 8) 134
6.5 Allocations for Process Fragment (Processes 9 - 16) 135
6.6 Planning Scenario 2: Infrastructure Deployment 143
6.7 Planning Scenario 2: Evaluation Results 146
6.8 Enforcement Scenario 1: Evaluation Results 149
6.9 Enforcement Scenario 2, Process 1: Evaluation Results 152
6.10 Enforcement Scenario 2, Process 2: Evaluation Results 153
6.11 Enforcement Scenario 2, Process 3: Evaluation Results 154

1

Introduction

Service-Oriented Architectures [Erl05], or SOA for short, have become widely adopted both in industry and research environments: standardized messages and message exchange formats such as WSDL and SOAP facilitate loose coupling, thus enabling service consumers and providers to interact in a much more flexible fashion than previously. One particularly interesting aspect of these SOAs is the possibility to combine several services into workflows (also known as "programming in the large").

Beyond the pure provisioning (or using) of functionality, however, both service providers and consumers usually have other interests: providers will strive for the best possible usage of their provided resources in order to maximize profit; conversely, consumers may want to execute an entire workflow as fast as possible, or as cheap as possible (or combinations thereof).

Consider the sample workflow given in Figure 1.1, which is a simplified version of an actual scientific workflow presented in [DGR+05] and is used for producing weather forecasts. Reasonable non-functional criteria that an end user might specify for the execution (of the entire workflow) could be "as fast as possible", or "as cheap as possible, but with a deadline so that the results are available for the evening news". All of the operations are available as Web Services and may be provided by one or more operation

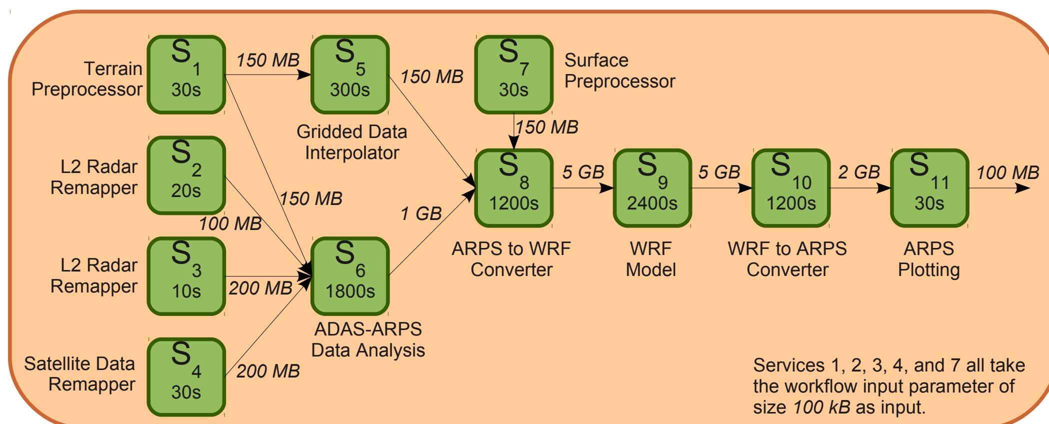


Figure 1.1: Weather Forecast Workflow

providers. Suppose that details on timing and data quantities of individual operations are as indicated in Figure 1.1. This implies that the overall execution of a single instance of this workflow is in the range of several hours – the exact duration strongly depends on the available resources. We therefore consider that workflow as a good example of a Scientific Workflow, as it is characterized by large volumes of data and contains long-running, CPU-intensive operations [Pla07, SPG06].

When multiple independent users invoke operations concurrently, these users are generally competing for the limited resources that providers have available. To meet non-functional requirements such as the ones mentioned above, thus being able to provide Quality of Service (QoS) guarantees to individual end users, resource access must happen in a controlled manner.

This thesis introduces a Workflow System termed DWARFS (Distributed Workflow system with Advance Reservation Functionality Support), which is capable of proactively controlling resource usage by leveraging Advance Resource Reservations (AR). The remainder of this document is organized as follows:

In Chapter 2, we introduce a formal model which defines the concepts and operations required to represent a system where service providers are aware of the resource requirements of the operations they make available, and where (planned) workflow executions are adapted to the state of the infrastructure.

Chapter 3 discusses our approach to planning workflow executions according to user-defined QoS criteria, while Chapter 4 focuses on the actual enforcement of reservations, i.e., on how the system can ensure that resource reservations are abided to. Chapter 5 presents the organization and interaction of the various services which constitute a DWARFS infrastructure, and in Chapter 6, we present various evaluations of both the planning and enforcement components.

In Chapter 7, we give an overview of related work. Finally, we conclude in Chapter 8 with a summary and a description of possible future research areas.

2

System Model

The purpose of this chapter is to formally define the model that serves as the foundation of the DWARFS system. This model spans many abstraction levels, ranging from individual CPU cycles to notions at the level of an entire network infrastructure. Ultimately, all the introduced concepts are interrelated and somewhat depending on each other. Therefore, this chapter starts by presenting a big picture of the system, which is gradually elaborated in more detail – the first part (Section 2.1) is meant to introduce key terms and concepts, and their relationships, in a way that is informal and easy to follow. Subsequently, the actual formal definitions are provided in a bottom-up manner in Sections 2.2 – 2.7. Section 2.8 concludes this chapter with a short summary of the model's core concepts, as well as discussions on some of its noteworthy aspects.

2.1 DWARFS at a glance

2.1.1 Scientific Workflows

The purpose of DWARFS is to plan and execute *Scientific Workflows*, or workflows for short. In its simplest and most abstract form as shown in Figure 2.1, a workflow is a composition of *activities*, and is typically depicted as a graph. The nodes of the graph represent the activities, whereas the edges, generally speaking, define dependencies between the activities. We will use the workflow depicted in Figure 2.1 as a running example throughout this chapter.

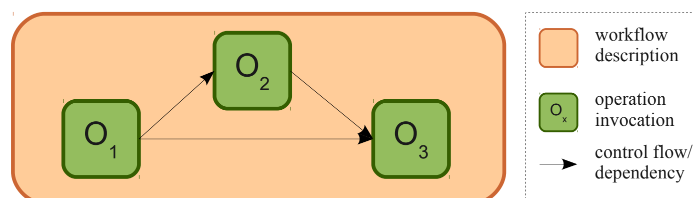


Figure 2.1: Sample workflow (abstract)

Each activity provides some kind of functionality. This functionality usually, though not necessarily, requires some input data, and produces some output data. Activities thus correspond to *operations* which, given some input, produce some output. Therefore, the edges of a workflow graph actually represent two distinct types of *dependencies*:

- Data flow: some or all of the input of an operation depends on some or all of the output of another operation.
- Control flow: an operation must not be executed before another operation has finished.

DWARFS is designed for a discrete operational model, where both input and output data are finite. In other words, all input data for an operation invocation is fully available at the time the operation is invoked, and an invocation ends when it has produced all its output. This results in an overlap, or rather inclusion, of the abovementioned dependency types: even if control flow is not explicitly specified, it is determined by the data flow, which yields a temporal dependency between the invocation of operations.

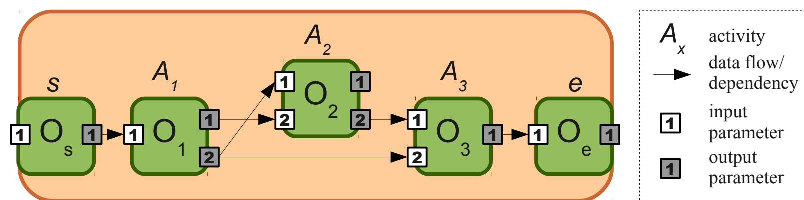


Figure 2.2: Sample workflow, with logical data flow

Figure 2.2 shows the same workflow as Figure 2.1, but at a slightly higher level of detail: it depicts the individual inputs and outputs of the operations contained within the workflow, as well as of the workflow itself. This figure also depicts a fundamental property of workflows: in fact, functionally, a workflow is itself an operation, producing outputs according to its inputs. Note that Figure 2.2 depicts two additional operations (O_s and O_e) when compared to Figure 2.1. In short, these operations represent the entry and exit points of the workflow itself, and will be discussed subsequently.

2.1.2 Resources, Hosts, Operation Providers

The mere availability of an *Operation Provider* in an SOA for any given *Operation* at least implies that "somewhere, there is a computer that is able to receive the input, process it, and return the output". In the simplest case, this means that somewhere in the world, a computer processor (CPU) will execute a few instructions in order to fulfill the request. In an SWF setting, this is more likely to be "some computer(s) will spend a considerable amount of effort for a considerable amount of time in order to process a large data input and produce a large data output". What is informally introduced as "effort" here actually corresponds to physical *Resources*, such as CPU, Random Access Memory (RAM) or intermediate storage, such as a local hard disk. Before and after the actual calculation, large amounts of data may need to be transmitted over the network.

Resources, such as CPU, storage, or bandwidth, are by their nature limited: they can only hold or process a limited amount of information (their *capacity*) at any given time. They are also bound to a physical component, i.e. the computer's processor, its hard disk, its network card, etc. A *Resource Provider* is an entity that controls one or more such components. The most typical example is a computer, or *Endpoint*, having associated CPU, storage, and networking resources. Just like in the real world, resources do not come for free: resource providers themselves are paying for their provision and maintenance, so when such resources are made available for clients to benefit from, the clients are charged for the usage. The *cost* for using resources is determined by *cost functions* defined by the resource's provider.

A single Resource Provider may be able to offer one or more Operations, thus acting as Operation Provider for multiple Operations. In fact, it can even offer multiple *Operation Instances* for the same Operation, such as using different algorithms for the same functionality (e.g., provide Bubble sort, Heap sort and Quick sort algorithms to perform a sort operation – all of which require different amounts of resources to execute).

While the declaration of the ability to provide an operation is an important part, actually providing it when it is requested is the other, most important, one. In a simple setting, an Operation Provider offers its services, and responds to all requests in a best-effort manner. As each request uses some of the resources required for the provisioning of the functionality, in a situation where the service becomes highly popular, the required resources get overloaded, thus deteriorating response times and QoS for all requests.

In the setting proposed in this work, QoS is achieved by managing resource utilization not reactively, but proactively: resources can only be utilized if they have been claimed beforehand. This in turn means that each individual operation invocation within a workflow execution is foreseen and scheduled before being carried out.

In a nutshell, before actually executing a workflow, DWARFS determines which resources need to be allocated where and when, and reserves these resources. At execution time, the reservations that have been made are leveraged, thus yielding a more predictable execution.

2.1.3 Resources and Reservations

Clearly, the underpinning foundation of DWARFS, required in order to provide predictable workflow execution, is resource management. We have seen that resource requirements need to be determined during planning, ahead of workflow execution. Of course, in order to actually execute a workflow according to its plan, the resource allocations need to be reserved by the corresponding resource providers.

In other words, there is a subtle yet important difference between the terms *Resource Allocation* and *Resource Reservation*: by the former term, we denote a prospected or actual utilization of the resource, while the latter represents an actual provider-side commitment to make the resources available. Put simply, only after all prospected resource allocations have actually been reserved with their respective providers can workflow planning be considered successful, and workflow execution begin.

While this section gave a short and informal overview of the key concepts in DWARFS, the following sections of this chapter will provide more formal definitions of the individual aspects, and show how exactly the introduced concepts are defined and relate to each other.

2.2 Operations and Workflows

Operations are one of the fundamental pillars of any calculation, and can in principle be understood as functions which, given some input data, deterministically produce some output data. In a sense, workflows are simply the composition of operations, thus acting as an operation themselves. In order to produce their output, operations need to be invoked to provide their functionality. In DWARFS, a single operation can be offered for provision by multiple entities.

2.2.1 Data Types

In principle, DWARFS does not impose any restrictions on the kinds of data that can be handled. However, for the sake of clarity, and because it is more suitable for formal verification, we use the concept of data types which allows for a more fine-grained classification of data.

The universe of data types is termed $\textsc{Type}$. □

2.2.2 Operation

At the highest level of abstraction, an operation can simply be understood as a procedure with produces output data from input data. We further specify how many inputs, and of which data types, an operation requires, and how many outputs of which types it produces. This is intentionally very similar to the operation signatures found in many programming languages.

Definition 2.1. *Operation*
An operation o is a tuple $o = (I, O, \phi)$, where:

- $I \in \textsc{Type}^n = (i_1, ..., i_n)$ is a tuple specifying the types of the operation's input parameters
- $O \in \textsc{Type}^m = (o_1, ..., o_m)$ is a tuple specifying the types of the operation's output parameters
- ϕ is a function $\phi : i_1 \times ... \times i_n \rightarrow o_1 \times ... \times o_m$, representing the actual functionality of the operation.

The universe of all operations is termed $\textsc{Op}$. □

As a trivial example, an operation which divides two natural numbers and returns a real number could be represented as $((\mathbb{N},\mathbb{N}),(\mathbb{R}),(a,b)\mapsto\frac{a}{b})$.

Note that operations may take no input ($I = ()$), or produce no output ($O = ()$). An example for the former might be a random number generator, an example for the latter an operation which prints its input to a physical printer. Such operations could depend on, or modify, some external state which is not captured in the model – in other words, they may have side effects. We explicitly allow for such side effects, as long as they are orthogonal to the model.

2.2.3 Workflows

In the spirit of "programming in the large", a workflow provides new functionality by re-using and recombining existing functionalities. This is achieved by composing operations into a graph, as shown in detail in Figure 2.2.

This figure illustrates a common, simple yet powerful, approach to defining workflows as a directed acyclic multigraph, where nodes depict the activities (or operations), and edges depict data flow. While we will go into further detail when giving the formal definition, there are several general characteristics of a workflow, most of which can be observed in the figure. First and foremost, a workflow as a whole is itself again an operation, taking inputs and producing outputs. Second, each contained operation's inputs, as well as the output of the workflow itself, are mapped from some previous contained operation's output, or the workflow's input. Third, because we consider individual data input and output parameters, the graph is in fact a multigraph, as it allows for multiple edges between nodes (representing multiple parameter transfers).

Note that the graph depicts the data flow dependencies inside the workflow (e.g., in the sample figure, the first output of operation O_1 is used as the second input of O_2; its second output is used as the first input of O_2, and as the second input of O_3). Such data flow dependencies necessarily imply control flow dependencies as well – i.e., at runtime, O_2 and O_3 cannot be executed before O_1 has produced its output. It may be necessary to define additional control flow, i.e., to merely state that some activity must not be started before another has ended, even if there is no direct or indirect data dependency.

Definition 2.2. *Workflow Description*
A workflow description wd is a tuple $wd = (I, O, A, \omega, \kappa, \delta)$, where:

- $I \in \textsc{Type}^n = (i_1, ..., i_n)$ is a tuple specifying the types of the workflow's input parameters
- $O \in \textsc{Type}^m = (o_1, ..., o_m)$ is a tuple specifying the types of the workflow's output parameters
- A is a non-empty set of activities corresponding to operation invocations. It contains at least two activities s and e, which denote the start and end of the workflow.
- ω is an injective mapping function $\omega : A \to \textsc{Op}$, associating each activity with an operation. The mappings for s and e are predefined such that $\omega(s) := o_s, \omega(e) := o_e$, where $o_s := (I, I, \text{id}), o_e := (O, O, \text{id})$.

- $\kappa \subseteq A \setminus \{e\} \times A \setminus \{s\}$ is a relation representing control flow edges of the workflow.
- $\delta \subseteq A \setminus \{e\} \times \mathbb{N}_+ \times A \setminus \{s\} \times \mathbb{N}_+$ is a relation representing data flow edges of the workflow.

and the following holds: $\forall (a_p, i, a_s, j) \in \delta \; \exists (a_p, a_s) \in \kappa$.

The universe of Workflow descriptions is termed WFD. □

Note that, as mentioned earlier, a workflow description $wd = (I, O, A, \omega, \kappa, \delta)$ can also be understood as an operation (I, O, ϕ), where ϕ is defined based on the definition of wd. Let us denote this mapping as the function $wfOp$: wfOp : WFD → OP.

While the start and end activities (s and e) may seem redundant at first glance, they are not: they provide single entry and exit points for the workflow (some practical impacts are explained further on). More importantly, within a workflow, they allow to "scatter" input data multiple times, using the data flow edges (δ), and to "gather" output in the same manner.

Let us briefly take a closer look at how control and data flow edges are represented by κ and δ. The existence of $(a_p, a_s) \in \kappa$ simply signifies that a control flow dependency exists between a_p and a_s, i.e., that a_s cannot be executed before a_p has finished. Data flow is defined in a very similar way, except that it additionally includes the information about which output is mapped to which input: $(a_p, i, a_s, j) \in \delta$ means that the i-th output of a_p constitutes the j-th input of a_s.

The following definitions partly use the fact that by the above definition, all data flow is also represented as control flow.

Definition 2.3. *Workflow Activity Precedence*
Let $wd = (I, O, A, \omega, \kappa, \delta)$ be a workflow description.

An activity $a_p \in A$ **directly precedes** an activity $a_s \in A$, denoted as $a_p \rightarrow a_s$, iff $(a_p, a_s) \in \kappa$. The **precedence** relation, denoted as $\xrightarrow{*}$, is the transitive closure of $\rightarrow$ over A. □

In the following, we may use the words "predecessor","successor","direct successor", etc. in the usual sense as one would expect from this definition.

Given the above definitions, it is still possible to define invalid workflows. Informally, a workflow is valid if its control flow graph is indeed a directed acyclic graph starting at the start activity and ending at the end activity, if all required operation inputs are assigned exactly once, and if there are no incompatible assignments (in terms of data types) in the data flow.

Definition 2.4. *Workflow Description Validity*
A workflow description $wd = (I, O, A, \omega, \kappa, \delta)$ is said to be **valid**, if and only if all of the following hold:

- $\forall a \in A \setminus \{s\} : s \xrightarrow{*} a$
- $\forall a \in A \setminus \{e\} : a \xrightarrow{*} e$

- $\nexists a \in A \mid a \xrightarrow{*} a$
- $\forall a \in A \setminus \{s\}, i \in [1, |\omega(a).I|] : \exists!(a_p, o_p, a, i) \in \delta$
- $\forall (a_p, o_p, a_s, i_s) \in \delta : \omega(a_p).O[o_p] = \omega(a_s).I[i_s]$

□

From now on, except when explicitly noted, we shall assume that all workflow descriptions referred to are valid.

Notational conventions

The previous definition uses shorthand notations which are meant to enhance readability, and which will be used occasionally throughout this document. We will shortly introduce these notations here, using the above example:

- **Element naming:** Similarly to the common notation in object-oriented languages, we use a dot-notation to refer to individual elements of tuples by the names used in their definition. In the above definition, $\omega(a).I$ thus refers to the input parameters of the operation that $\omega(a)$ denotes.
- **Tuple cardinality:** When referring to the number of elements of tuples of arbitrary size, we use the notation $|tuple|$, in analogy to set cardinalities. In the definition above, this notation is used in the expression $|\omega(a).I|$.
- **Tuple index:** To directly address a given element of a tuple of arbitrary size, we use brackets, as commonly used in programming languages to address array elements. Thus, $\omega(a_p).O[o_p]$ refers to the o_p^{th} element of the tuple of outputs of the operation denoted by $\omega(a_p)$.

As said, these shorthand notations are introduced to provide a more intuitive and readable representation, and are typically used in conjunction, as seen above.

2.3 Resources

The provision of any kind of functionality naturally requires resources in order to be fulfilled. The Oxford English Dictionary defines resources as "stocks or reserves of money, materials, people, or some other asset, which can be drawn on when necessary" [Oxf10]. Another definition states that "A resource is a source or supply from which benefit is produced. Typically resources are materials, services, staff, or other assets that are transformed to produce benefit and in the process may be consumed or made unavailable."[Res13]

Informally, resources (and their limited availability) are ubiquituously affecting all kinds of computer usage: most users have probably already been affected by full disks or programs crashing due to insufficient Random Access Memory (RAM). Likewise, computations may be painfully slow on old computers, the duration for completing

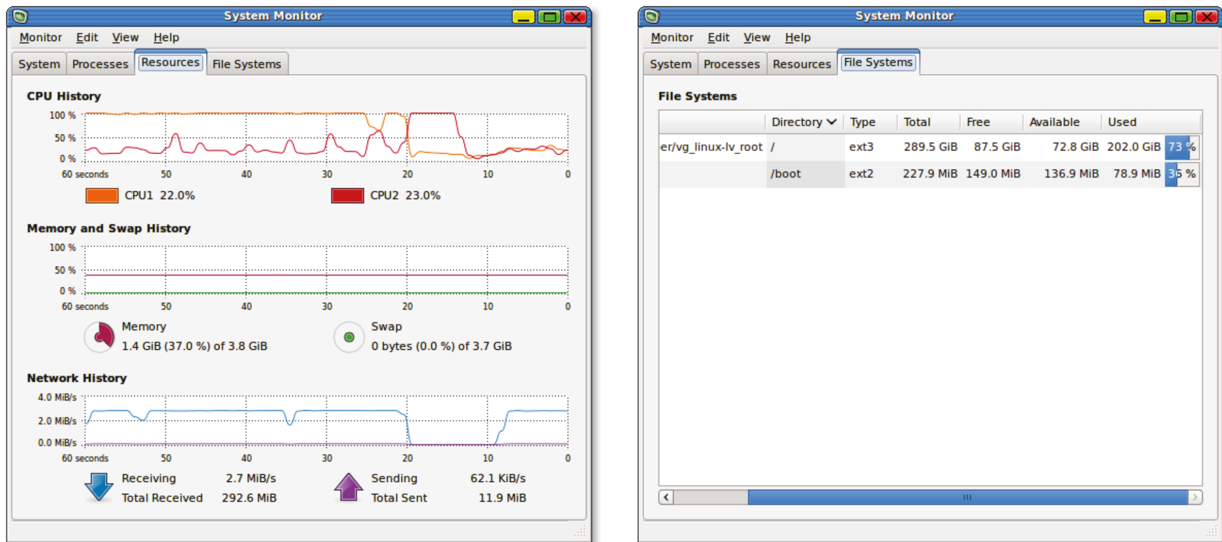


Figure 2.3: Screenshots: Resource Monitoring

a download depends on the capacity of the connection, etc. Figure 2.3 shows examples of how the usage of various resources can be represented and monitored using Ubuntu 10.04.

One of the most important aspects of the DWARFS system is that it is aware of the ramifications of the limited availability of resources. In fact, resources are the most fundamental concept that underpins the entire system.

2.3.1 Resource classes, types, and instances

We have already informally introduced various resources, such as computing power, storage, and bandwidth. Taking a closer look at their intended purposes, it becomes clear that there are in fact two radically distinct kinds of resources, which can be classified according to their behavior. Figure 2.4 shows an intuitive example outside of the Computer Science world: a barrel, and a hose. Both of these items can be considered resources "usable" with water.

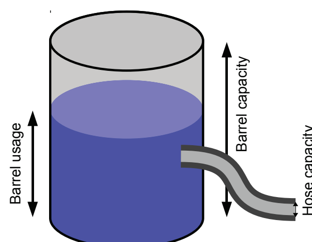


Figure 2.4: Example of Resource Classes

The barrel can hold water up to its capacity, while the hose can discharge water up to its capacity. In other words: the barrel has *container* semantics, i.e., its capacity

determines how much water it *persistently* holds, while the hose has *throughput* semantics: its capacity determines the maximal *transitory* throughput of water. The following definition formalizes this classification.

Definition 2.5. *Resource Classes*
The universe of resource classes considered in DWARFS is defined as follows:

$$\textsc{ResClass} = \{\textsc{Persistent}, \textsc{Transitory}\}.$$

□

Specifically, the two classes have the following meanings:

- The class of persistent resource types PERSISTENT encompasses all resources which have container semantics.
- The class of transitory resource types TRANSITORY encompasses all resources which have throughput semantics.

While resource classes provide a distinction based on intrinsic *behavioral* properties, we also want to distinguish resources based on their *purpose*. For example, while a processor, a hard disk, and a network card could be classified solely as representing persistent or transitory resources, a more fine-grained distinction into several *Resource Types* (e.g., CPU,STORAGE,BANDWIDTH) – makes sense, and is assumed within our model.

Definition 2.6. *Resource Type*
A Resource Type represents an abstract notion of a family of devices or features, required to perform a specific functionality. The universe of all resource types is named RESTYPE. We assume the existence of a mapping function, which associates a resource type with the corresponding resource class:

$$\text{resClass} : \textsc{ResType} \rightarrow \textsc{ResClass}$$

□

While Resource Types denote functionalities, they are not associated with any physical entity by themselves. An actual *Resource* is a concrete instance of a given resource type, with a limited, discrete capacity. Resources are managed by their owner and can generally be shared between several consumers, up to their capacity limit.

Definition 2.7. *Resource*
A Resource r is a tuple $r = (t, c)$, where:

- $t \in \textsc{ResType}$ is the resource's resource type.
- $c \in \mathbb{N}_+$ is the capacity of the resource, in a unit not further specified, but conforming to the semantics of its resource type and class.

The universe of all resources is named RESOURCE. □

2.3.2 Common resource types

There are a number of resource types which are considered so fundamental that DWARFS includes predefined definitions for them. As the purpose of DWARFS is to execute computationally expensive and data-intensive workflows in a distributed infrastructure, it is natural to apply this to the following resource types.

Computational Power: CPU

Every computing device must have at least one working Central Processing Unit (CPU) in order to be functional. We define the CPU resource of a computer as comprising all available physical CPUs. Of course, this resource is being used by every calculation performed on that computer.

The capacity of a resource of this type can roughly be understood as its "compute power". Suitable units for the capacity might be "Computations per second", however it is up to the resource's owner to specify the exact values. Formally, this resource type is specified as:

$\textsc{Cpu} \in \textsc{ResType} \mid \text{resClass}(\textsc{Cpu}) = \textsc{Transitory}$

Storage

When processing large amounts of data, it is practically inevitable to temporarily store this data on secondary storage (i.e., hard disk). The capacity of a resource of this type corresponds to the total attached amount of secondary storage; a natural unit for such a resource is the number of available bytes. More formally:

$\textsc{Storage} \in \textsc{ResType} \mid \text{resClass}(\textsc{Storage}) = \textsc{Persistent}$

Bandwidth

In a distributed system, any communication requires network connectivity. Transfer speed is impacted by the availability of network resources (bandwidth) of all parties involved in the transfer. As a network transfer always involves at least two parties, the units for resources of this type must be compatible. Capacities for network resources are thus always expected to correspond to the maximum net throughput, for instance in bytes per second.

$\textsc{Bandwidth} \in \textsc{ResType} \mid \text{resClass}(\textsc{Bandwidth}) = \textsc{Transitory}$

2.3.3 Other Resource Types

DWARFS is by no means limited to the above resource types. In fact, virtually anything that meets the few criteria for qualifying as a resource and that is required to properly execute a workflow can be considered by DWARFS.

For example, if an organization provides an operation that requires access to a telescope, one could define an appropriate resource type mapped to the PERSISTENT class. The capacity of the concrete resource could reflect the number of attached telescopes, and each invocation of an operation would need to access / reserve one or more of them.

2.4 Resource Usage

The physical presence of resources alone is of little interest when they are not used. Conversely, if a resource had an unlimited capacity, there would be no need for it to be managed at all, as it would be available at all times to everyone requesting to use it. Clearly, the main interest thus lies in the management of such limited resources. Proper resource management implies knowing when, and to which extent, resources are being used.

Before we go into details on how resource usage can be represented, let us shortly discuss how resources relate to each other. From the previous definitions of resources it should be apparent that resources are generally not "standalone" entities; rather, multiple resources can belong to the same (physical) entity managing them – for example, a computer is managing its own CPU, storage, bandwidth and associated peripherals. We call such a grouping of resources *Resource Provider*.

Definition 2.8. *Resource Provider*
A Resource Provider rp is an entity which provides and manages one or more resources. Formally, it is defined as the set of resources that it manages: $rp \subseteq$ RESOURCE.
The universe of all resource providers is termed RESPROV. □

Note that resource providers have disjoint sets of associated resources, i.e., no two resource providers manage the same resource: $\forall rp_1, rp_2 \in$ RESPROV $: rp_1 \neq rp_2 \Rightarrow rp_1 \cap rp_2 = \emptyset$.

2.4.1 Usage Blocks

A usage block represents the constant and continuous, intended or actual usage of a certain amount of some (any) resource. Note that we are explicitly not binding usage blocks to any particular resource.

Definition 2.9. *Usage Block*
A Usage block u is defined as a tuple $u = (s, e, a)$, where:

- $s \in \mathbb{N}$ is the start timestamp, considered inclusive
- $e \in \mathbb{N}$ is the end timestamp, considered exclusive
- $a \in \mathbb{N}$ is an amount of usage

and the following holds:

- $s < e$

The universe of all usage blocks is named USAGE. □

The reason for not binding usage blocks to any particular resource is that in fact, usage blocks are a simple "helper" construct that will ease the subsequent definition of allocations (which are expressed using such usage blocks). To formulate an analogy with measurands – usage blocks allow to express individual values independent of the unit (the resource in this case), but it is the combination of values and unit which forms the object of interest (allocations in this case). For a pragmatic example, keeping usage blocks as "basic units" allows to easily express situations where one needs to acquire the same set of usage blocks for multiple resources (Section 2.4.3).

However, of course not all resources have the same capacity, and therefore not all of them can accommodate all usage blocks. The following definition captures this fact.

Definition 2.10. *Usage Block Validity*
A usage block $u = (s, e, a)$ is said to be **valid** for a resource $r = (t, c)$, iff $a \leqslant c$. □

2.4.2 Allocations

Allocations represent the intended or actual usage of one particular resource over time, represented as a combination of usage blocks. An allocation must contain zero or more non-overlapping usage blocks.

Definition 2.11. *Allocation*
An allocation a is defined as a tuple $a = (U, r)$, where:

- $U \subseteq$ USAGE is a set of usage blocks
- $r \in$ RESOURCE is a resource

and the following holds:

- $\nexists (s_1, e_1, a_1), (s_2, e_2, a_2) \in U : s_1 < e_2 \wedge s_2 < e_1$.

The universe of all allocations is named ALLOC. We further define the following functions on allocations:

$$\begin{aligned}
\text{allocStart} : \textsc{Alloc} &\to \mathbb{N} \\
(U, r) &\mapsto \begin{cases} 0 & \text{if } U = \varnothing \\ \min\limits_{(s_i, e_i, a_i) \in U} (s_i) & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\text{allocEnd} : \textsc{Alloc} &\to \mathbb{N} \\
(U, r) &\mapsto \begin{cases} 0 & \text{if } U = \varnothing \\ \max\limits_{(s_i, e_i, a_i) \in U} (e_i) & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\text{allocAmount} : \textsc{Alloc} \times \mathbb{N} &\to \mathbb{N} \\
((U, r), t) &\mapsto \begin{cases} a & \text{if } \exists (s, e, a) \in U \mid s \leqslant t < e \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

□

Definition 2.12. *Allocation equivalence*
Two allocations $a_1 = (U_1, r_1), a_2 = (U_2, r_2)$ are said to be **equivalent**, denoted as $a_1 \triangleq a_2$, if and only if:
$r_1 = r_2 \wedge \forall t \in \mathbb{N} : \text{allocAmount}(a_1, t) = \text{allocAmount}(a_2, t)$ □

The definitions of allocations and their equivalence suggest that there is the possibility to express the same "fact" using various allocations. For a simple example, let r be a resource. Then, the following allocations are all equivalent and in fact correspond to an empty allocation: $(\varnothing, r) \triangleq (\{(0,1,0)\}, r) \triangleq (\{(15,23,0),(42,65,0)\}, r)$.

Therefore, we define and use a canonical representation of allocations. Informally, the conditions for canonicity of an allocation are that it is contiguous (i.e., there are no gaps between the usage blocks), that two immediately succeeding usage blocks do not share the same amount, and that it does not begin or end with a usage block of amount 0.

Definition 2.13. *Allocation canonicity*
An allocation $a = (U, r)$ is said to be **canonical** if and only if all of the following hold:

- $\forall (s_i, e_i, a_i) \in U \mid e_i \neq \text{allocEnd}(a) : \exists (s_j, e_j, a_j) \in U : s_j = e_i \wedge a_j \neq a_i$
- $\nexists (s_s, e_s, a_s) \in U \mid s_s = \text{allocStart}(a) \wedge a_s = 0$
- $\nexists (s_e, e_e, a_e) \in U \mid e_e = \text{allocEnd}(a) \wedge a_s = 0$

□

Note that for any allocation, we can construct an equivalent canonical allocation by applying Algorithm 1.

Algorithm 1: Determine equivalent canonical allocation

Input: An allocation $a = (U, r)$
Output: A canonical allocation $c : a \triangleq c$
$\mathfrak{U} \Leftarrow U$;
while $\exists (s_0, e_0, 0) \in \mathfrak{U}$ **do**
```
// Remove all usages of amount zero
```
$\mathfrak{U} \Leftarrow \mathfrak{U} \setminus (s_0, e_0, 0)$;
while $\exists (s_1, e_1, a_1), (s_2, e_2, a_2) \in \mathfrak{U} \mid e_1 = s_2 \wedge a_1 = a_2$ **do**
```
// Join successive usages with same amount
```
$\mathfrak{U} \Leftarrow (\mathfrak{U} \setminus \{(s_1, e_1, a_1), (s_2, e_2, a_2)\}) \cup \{(s_1, e_2, a_2)\}$;
while $\exists (s_1, e_1, a_1), (s_2, e_2, a_2) \in \mathfrak{U} \mid e_1 < s_2 \wedge s_2 = \min_{(s_i, e_i, a_i) \in \mathfrak{U} : s_i \geqslant e_1} (s_i)$ **do**
```
// Fill inner gaps between usages with usages of amount 0
```
$\mathfrak{U} \Leftarrow \mathfrak{U} \cup \{(e_1, s_2, 0)\}$
return $(\mathfrak{U}, r)$

Definition 2.14. *Allocation validity*

An allocation $a = (U, r)$ is said to be **valid** iff a is canonical and $\forall u_i \in U : u_i$ is valid for r. □

Lemma 2.15. *Existence and uniqueness of equivalent canonical allocation*

For every allocation there exists exactly one equivalent canonical allocation.
Proof:

Existence:

For any given allocation, Algorithm 1 allows to determine an equivalent canonical allocation.

Uniqueness:

Suppose, for the sake of contradiction, that there are two allocations, $a_1 = (U_1, r)$ and $a_2 = (U_2, r)$, and that $a_1 \triangleq a_2$, and both a_1 and a_2 are canonical.

$$\begin{aligned} a_1 \neq a_2 \Rightarrow U_1 \neq U_2 & \\ & \Leftrightarrow \exists (s_x, e_x, a_x) \in U_1 \mid (s_x, e_x, a_x) \notin U_2 \vee \\ & \quad \exists (s_y, e_y, a_y) \in U_2 \mid (s_y, e_y, a_y) \notin U_1 \end{aligned}$$

Without loss of generality, we consider only the first case (the second is symmetric). Let t be any timestamp encompassed by (s_x, e_x, a_x), i.e., $s_x \leqslant t < e_x$.

$$\begin{aligned} a_1 \triangleq a_2 &\Rightarrow \text{allocAmount}(a_1, t) = \text{allocAmount}(a_2, t) = a_x \\ &\Leftrightarrow a_x = 0 \vee \exists (s_t, e_t, a_x) \in U_2 : s_t \leqslant t < e_t \end{aligned}$$

In other words: for any timestamp t encompassed by (s_x, e_x, a_x), there must exists a usage encompassing t and with amount a_x in a_2 (case b), except possibly in the simple case where $a_x = 0$ (case a).

case a) $s_x \leqslant t < e_x \wedge a_x = 0 \wedge \nexists (s_t, e_t, a_x) \in U_2 : s_t \leqslant t < e_t$:

$$\begin{aligned} \text{if } e_x \leqslant \text{allocStart}(a_2) &\Rightarrow a_1 \text{ is not canonical, because it contains leading} \\ &\quad\ \text{usages of amount 0.} \ \lightning \\ \text{if } s_x \geqslant \text{allocEnd}(a_2) &\Rightarrow a_1 \text{ is not canonical, because it contains trailing} \\ &\quad\ \text{usages of amount 0.} \ \lightning \\ \text{in all other cases} &\Rightarrow a_2 \text{ is not canonical, because its usages are not} \\ &\quad\ \text{contiguous.} \ \lightning \end{aligned}$$

case b) $s_x \leqslant t < e_x \wedge \exists (s_t, e_t, a_x) \in U_2 : s_t \leqslant t < e_t$:

$$\begin{aligned} (s_x, e_x, a_x) \notin U_2 &\Leftrightarrow \neg (s_x = s_t \wedge e_x = e_t) \\ &\Rightarrow \text{at least one of the following applies:} \\ \text{if } s_t < s_x &\Rightarrow \exists (s_f, e_f, a_x) \in U_1 : e_f = s_x \\ &\Rightarrow a_1 \text{ is not canonical, because it contains consecutive} \\ &\quad\ \text{usages of the same amount.} \ \lightning \\ \text{if } s_x < s_t &\Rightarrow \exists (s_f, e_f, a_x) \in U_2 : e_f = s_t \\ &\Rightarrow a_2 \text{ is not canonical, because it contains consecutive} \\ &\quad\ \text{usages of the same amount.} \ \lightning \\ \text{if } e_x < e_t &\Rightarrow \exists (s_f, e_f, a_x) \in U_1 : s_f = e_x \\ &\Rightarrow a_1 \text{ is not canonical, because it contains consecutive} \\ &\quad\ \text{usages of the same amount.} \ \lightning \\ \text{if } e_t < e_x &\Rightarrow \exists (s_f, e_f, a_x) \in U_2 : s_f = e_t \\ &\Rightarrow a_2 \text{ is not canonical, because it contains consecutive} \\ &\quad\ \text{usages of the same amount.} \ \lightning \end{aligned}$$

□

From here on, except if noted otherwise, we shall assume that all allocations referred to are canonical.

Allocation addition

When an allocation $a = (U, r)$ is valid, this simply indicates that resource r can, in principle, accommodate all usage blocks in U. As previously noted, resources can generally be shared – in other words, they can accommodate multiple allocations' usages. Instead of reasoning on sets of allocations, it is often convenient to aggregate all the contained allocations into a single one (by adding them). For example, a simple way to verify if a set of allocations can be accommodated is to verify whether the sum of its elements is valid. An example is shown in Figure 2.5, which depicts the addition of two valid allocations; the resulting allocation is invalid for a resource with capacity 100.

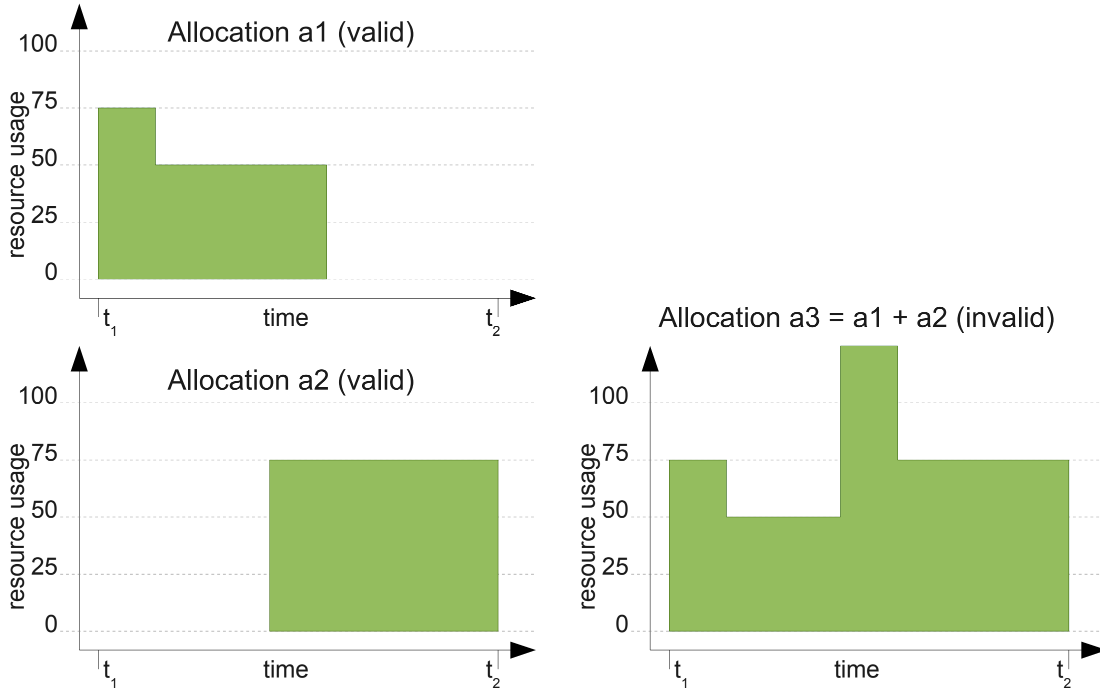


Figure 2.5: Allocation Addition Example

Definition 2.16. *Allocation addition*
Addition on allocations is a binary, associative and commutative operation defined by the following function:

$$\begin{aligned} \text{allocAdd} : \textsc{Alloc} \times \textsc{Alloc} &\to \textsc{Alloc} \\ ((U_1, r), (U_2, r)) &\mapsto (U_3, r) : \forall t \in \mathbb{N} : \text{allocAmount}((U_3, r), t) \\ &\qquad = \text{allocAmount}((U_1, r), t) + \text{allocAmount}((U_2, r), t) \wedge \\ &\qquad (U_3, r) \text{ is canonical.} \end{aligned}$$

□

We use the standard infix operator notation for addition, thus: $a_1 + a_2 \equiv \text{allocAdd}(a_1, a_2)$. Note that addition is only defined for allocations referring to the same resource.

2.4.3 Co-allocations: interdependent allocations

Generally, the utilization of some functionality (i.e., invoking an operation) requires joint allocations not a for single, but for multiple resources, with these resources possibly being owned by multiple independent parties. For example, as shown in Figure 2.6, a remote operation invocation will result in the transmission of input data from the caller to the callee (requiring bandwidth on both sides), then the actual execution of the request (requiring, for example, CPU and a telescope at the callee), and finally the transmission of the result (again requiring bandwidth on both sides). During the entire process, the callee may require storage capacity to hold the input and/or output data.

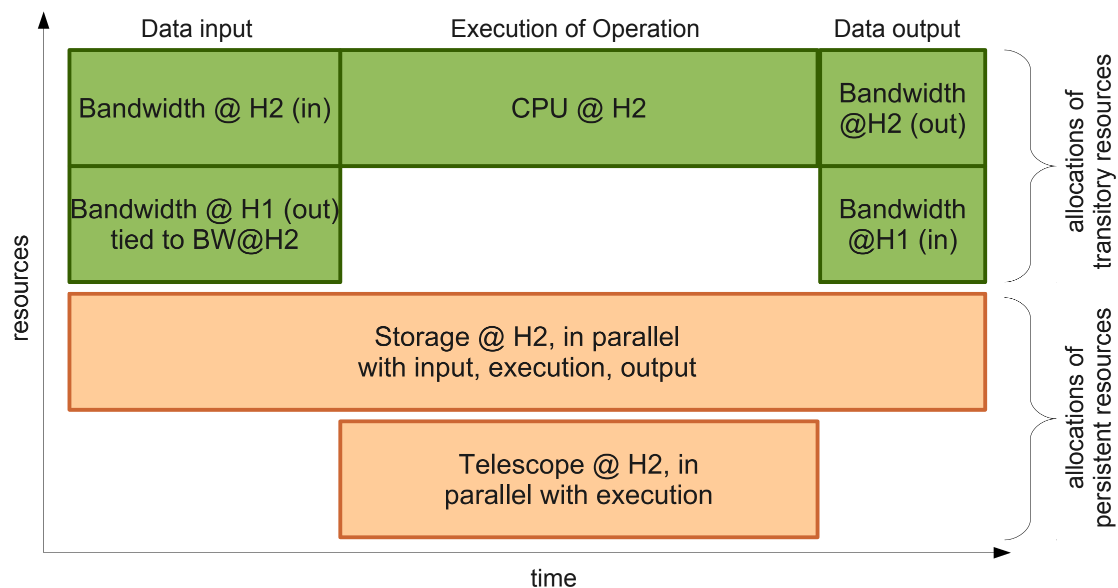


Figure 2.6: Sample Co-allocation for an operation invocation

While this invocation is – from the logical perspective – a single step, namely a single operation call, it is in fact making coordinated use of several distinct resources. Co-allocations capture this notion of coordinated use. Before we go into the formal definition, let us take a further look at some of their characteristics. There are two general, orthogonal patterns found in co-allocations:

Sequentiality

As implied by the name, this means that one resource is used (strictly) after another. In the example, the actual calculation (requiring CPU) can only be performed after data upload (requiring Bandwidth) is completed.

Parallelism

This applies when several resources are being used at the same time. A prime example for parallelism are network transfers, as they require bandwidth allocations (at least) on the sending and on the receiving end. This is again demonstrated by the above example.

Combinations

Dependencies among allocations are not limited to either sequentiality or parallelism, but typically are combinations of both. In the example, storage at the callee's side is allocated in parallel to a a sequence of other allocations. As we shall see in the following, not all combinations that are possible are also meaningful.

Definition 2.17. *Co-allocation*
A Co-allocation C represents the intended or actual usage of one or more resources over time. It is defined as a set of valid allocations: $C \subseteq \{a \in \textsc{Alloc} \mid a \text{ is valid}\}$

The universe of all co-allocations is named $\textsc{CoAlloc}$.

We further define a co-allocation's start and end timestamps as functions of the contained allocations:

$$\begin{aligned} \text{coAllocStart} : \textsc{CoAlloc} &\to \mathbb{N} \\ C &\mapsto \min_{a \in C}(\text{allocStart}(a)) \\ \text{coAllocEnd} : \textsc{CoAlloc} &\to \mathbb{N} \\ C &\mapsto \max_{a \in C}(\text{allocEnd}(a)) \end{aligned}$$

□

2.4.4 Allocation And Co-Allocation Cost

Allocations make use of resources, and resources must be held available. This ultimately leads to (real-world) running costs associated with the provisioning of the resources. It is thus natural to ask for a recompensation for resource use. Therefore, we assume the existence of a global cost function, which allows to determine the cost of an allocation.

Definition 2.18. *Allocation Cost*
The cost of an allocation is determined by the following function:

$$\text{allocCost} : \textsc{Alloc} \times \mathbb{N} \to \mathbb{R}_+$$

$\text{allocCost}(a, t)$ returns the (non-negative) cost of an allocation a, when requested at timestamp t. □

Just like in the real world, the cost of using a resource may depend on *when* it is requested: an allocation that is requested just before it starts (last minute) may be significantly cheaper than if it is still far in the future.

As co-allocations are merely sets of allocations, a naïve approach to determine the cost of a co-allocation would be to simply add the costs of all its allocations. However, consider the following real-world examples: Cloud providers generally charge for network transfers, but waive the costs if these transfers are entirely within their own network. In other words, "a co-allocation is more than the sum of its parts" in that it provides additional information which may lead to an adjustment of the costs.

Definition 2.19. *Co-allocation Cost*
The cost of a co-allocation is determined by the following function:

$$\text{coAllocCost} : \textsc{CoAlloc} \times \mathbb{N} \rightarrow \mathbb{R}_{+}$$

$\text{coAllocCost}(C, t)$ returns the cost of a co-allocation C, when requested at timestamp t. We place an additional restriction in that the cost may be lowered, but not raised:
$\text{coAllocCost}(C, t) \leqslant \sum_{a \in C} \text{allocCost}(a, t)$.

□

2.4.5 Co-Allocation Constraints

As described previously, co-allocations represent a set of allocations which depend on each other in some way. These dependencies implicitly manifest themselves by how the contained allocations are related to each other in terms of parallelism and sequentiality, i.e., in the co-allocation's "structure", but they are not made explicit in the co-allocation itself.

Consider again Figure 2.6, and the description of it given above. The description is actually anticipating in which way the individual parts of the co-allocation need to be (temporally) related, even if the exact allocations are not known yet. In other words, co-allocations merely represent the materialized results of some co-allocation *constraints*, whereby co-allocation constraints can be understood as blueprints determining the rules for co-allocations to abide.

Before going into further detail, there are a few important observations to make, concerning the behavior of resources according to their resource class.

- **Persistent Resources** have container semantics. A persistent resource can never accommodate more than its capacity. For example, a hard disk of 250 GB will not, at any moment in time, be able to store more than a total of 250 GB.

- **Transitory Resources** have throughput semantics. Their capacity refers to their ability to "work on" a particular amount *at a given time*. For example, while a network connection with capacity 64 kbit/s cannot accomodate more than 64 kbit at any moment, it is able to transfer an arbitrary amount of data, given enough time.

Sticking with the above concrete examples, a request to transfer a certain amount of data is very different from a request to temporarily hold that amount of data. In the former case, the time required for the transfer is determined by the amount itself, the capacity of the connection, and the usage of the connection. In the latter case, the time frame of the allocation is in principle not known at all, unless it is mandated in some other manner.

This pattern is generally valid, and also visible in Figure 2.6: allocations for persistent resources depend on one or more allocations for transitory resources, as only

allocations for transitory resources allow to determine for how long resources are actually required. This is further elaborated in the following. The subsequent definitions are provided in a "bottom-up" manner, considering constraints for the individual resource types first, before finally being combined to specify how co-allocation constraints can be expressed.

Time-Determining Allocation Constraints

A time-determining allocation constraint defines constraints for allocations for one or more resources of the same transitory resource type. It allows to specify how much of the affected resources is required in total, and upper and lower bounds on the usages at any time. For example, it can specify that a total amount of 2 GB must be transferred from some resource provider's outgoing network interface, to another resource provider's incoming network interface, at a throughput rate no lower than 2 MB/s and no higher than 200 MB/s.

Definition 2.20. *Time-determining Allocation Constraint*
A time-determining allocation constraint tc is defined as a tuple $tc = (R, min, max, sum)$, where

- $R \subseteq \textsc{Resource}$ is a non-empty set of resources of the same (transitory) resource type
- min is an amount of usage that must be provided at all times by each and every $r \in R$
- max is an amount of usage that must not be exceeded at any time, by any $r \in R$
- $sum \geqslant 0$ is the total resource usage that must be provided by each and every $r \in R$

where all of the following hold:

- $\forall r_1, r_2 \in R : r_1.t = r_2.t$
- $\forall r \in R : \text{resClass}(r.t) = \textsc{Transitory} \wedge r.c \geqslant max$
- $0 < min \leqslant max$

The universe of time-determining allocation constraints is termed $\textsc{Tac}$. □

As mentioned previously, allocation constraints are meant as "blueprints" for allocations. Informally, an allocation complies to that blueprint if it does not violate any of the constraints established by it. For a time-determining allocation constraint, that specifically means that the allocation concerns one of the respective resources, stays within the usage amount limits, and the total amount of all its usages matches or exceeds the requested one.

Definition 2.21. *Time-Determining Allocation Constraint Compliance*
An allocation a is said to **comply with** a time-determining allocation constraint tc, denoted as $a \xrightarrow{\checkmark} tc$, if and only if all of the following hold:

- a is valid
- $a.r \in tc.R$
- $\forall t \mid \text{allocStart}(a) \leqslant t < \text{allocEnd}(a) : tc.min \leqslant \text{allocAmount}(a, t) \leqslant tc.max$
- $tc.sum \leqslant \sum_{\text{allocStart}(a) \leqslant t < \text{allocEnd}(a)} \text{allocAmount}(a, t)$

□

Note that potentially many allocations can comply with a single time-determining allocation constraint. This statement is somewhat trivial, because the allocation constraint does not prescribe any "time frame" for the start or end of compliant allocations, and compliant allocations may actually use more resources than required (but not less). However, even if we fixed one, or both, of start and end timestamps, and used exactly the resources that the constraint prescribes, there could still be a large number of compliant allocations.

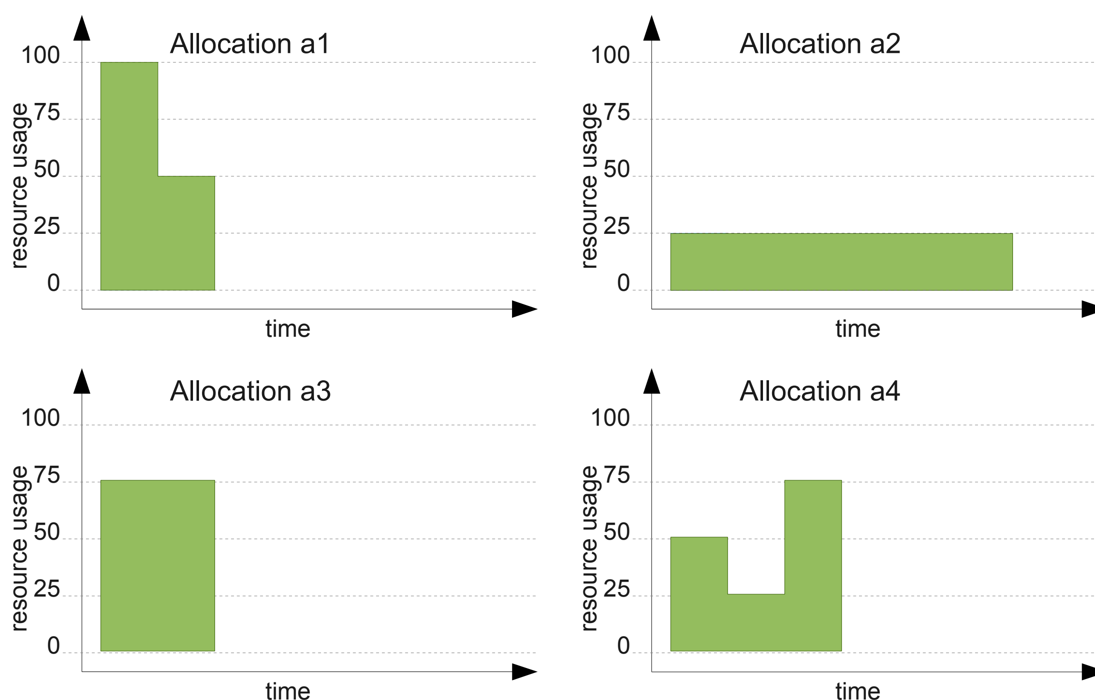


Figure 2.7: Alternative Allocations for a Time-Determining Allocation Constraint

This fact is graphically depicted in Figure 2.7. The figure shows several possible allocations for a transitory resource r with capacity 100, all of which are compliant with a time-determining allocation constraint $tc = (\{r\}, 25, 100, 150)$. Informally, it is the *area* encompassed by the allocations' usages that is equal, but the individual usage blocks may be different.

This property is also the reason why we termed time-determining allocation constraints this way: It is the compliant allocation itself which determines the time frame the resource needs to be allocated in order to fulfill the constraint.

In a sense, time-determining allocation constraints allow us to formalize restrictions for compliant allocations. Going one step further, we now consider the possibility to "refine" such allocation constraints, which allows us to further restrict the set of compliant allocations (for example, by reducing the interval allowed for minimum/maximum usage amount).

Definition 2.22. *Time-Determining Allocation Constraint Restriction*
A time-determining allocation constraint tc_r is said to **restrict** a time-determining allocation constraint tc_b, denoted as $tc_r \trianglelefteq tc_b$, if and only if:

- $tc_b.sum = tc_r.sum$
- $tc_b.R = tc_r.R$
- $tc_b.max \geqslant tc_r.max$
- $tc_b.min \leqslant tc_r.min$

□

Effectively, this means that any allocation compliant with tc_r is also compliant with tc_b: $\forall a \in \textsc{Alloc} : a \xrightarrow{\checkmark} tc_r \Rightarrow a \xrightarrow{\checkmark} tc_b$. The reverse does not necessarily hold.

Definition 2.23. *Time-Determining Allocation Constraint Augmentation*
A time-determining allocation constraint tc_a is said to **augment** a time-determining allocation constraint tc_b, if and only if:

- $tc_b.sum = tc_a.sum$
- $tc_b.R \subseteq tc_a.R$
- $tc_b.max = tc_a.max$
- $tc_b.min = tc_a.min$

□

Augmenting a time-determining allocation constraint corresponds to the inclusion of new resources into the set of resources that need to be allocated in order to fulfill the constraint. An intuitive example is a network transfer, where (the same usage blocks of) bandwidth must be allocated both on the sending and the receiving end, and possibly at intermediate network hops as well.

Let us now consider the combination of augmentation and restriction for time-determining co-allocation constraints. Such a combination enables us to "refine" time-determining allocation constraints by both including additional resources to be allocated as well as limiting the permissible range of resource usage amounts.

Definition 2.24. *Time-Determining Allocation Constraint Augmented Restriction*
A time-determining allocation constraint tc_α is said to be an augmented restriction of a time-determining allocation constraint tc_β, denoted as $tc_\alpha \dddot{\trianglelefteq} tc_\beta$, if and only if there exists a time-determining allocation constraint tc_χ such that $tc_\alpha \trianglelefteq tc_\chi$, and tc_χ augments tc_β. □

Intuitively, the notion of augmented restriction is most useful when reasoning over sets of allocations, where all elements of the set must comply with a time-determining allocation constraint – as is the case for co-allocation constraints, which we shall introduce soon.

Dependent Allocation Constraints

A dependent allocation constraint defines constraints for allocations for exactly one resoure of a persistent resource type. It allows to specify how much of the affected resource must be available at any given time.

Definition 2.25. *Dependent Allocation Constraint*
A dependent allocation constraint dc is defined as a tuple $dc = (r, min)$, where

- $r \in$ RESOURCE
- min is a usage amount (inclusive) that must be provided at all times

where all of the following hold:

- resClass$(r.t) =$ PERSISTENT
- $0 < min \leqslant r.c$

The universe of dependent allocation constraints is termed DAC. □

Just like for time-determining allocation constraints, we can determine whether an allocation complies with a dependent allocation constraint, i.e., whether it fulfills all of the requirements of the constraint.

Definition 2.26. *Dependent Allocation Constraint Compliance*
An allocation a is said to **comply with** with a dependent allocation constraint dc, denoted as $a \xrightarrow{\checkmark} dc$, if and only if all of the following hold:

- a is valid
- $a.r = dc.r$
- $\forall t \mid \text{allocStart}(a) \leqslant t < \text{allocEnd}(a) : dc.min \leqslant \text{allocAmount}(a, t)$

□

We again define the meaning of restriction when referring to dependent allocation constraints.

Definition 2.27. *Dependent Allocation Constraint Restriction*
A dependent allocation constraint r is said to **restrict** a dependent allocation constraint b, denoted as $r \trianglelefteq b$, if and only if:

- $b.res = r.res$
- $b.min \leqslant r.min$

□

Now that we have defined the basic (time-determining, and dependent) allocation constraints, we can move on to combine them into co-allocation constraints. A co-allocation constraint allows to define the interdependencies of basic allocation constraints, thus defining the structure of compliant co-allocations.

A co-allocation constraint consists of a number of temporally dependent (i.e., consecutive) time-determining allocation constraints, as well as a number of dependent allocation constraints, where dependent allocation constraints are associated with the time-determining allocation constraints that they start and end with.

Definition 2.28. *Co-Allocation Constraint*
A co-allocation constraint cc is a tuple $cc = (T, <, D, \tau)$, where:

- $T \subseteq$ TAC is a non-empty set of time-determining allocation constraints
- $<$ is an non-reflexive binary relation on T such that $(T, <)$ is a strict total order, indicating the temporal relationship between the time-determining allocation constraints.
- $D \subseteq$ DAC is a set of dependent allocation constraints
- $\tau \subseteq D \times T \times T$ is a ternary relation associating dependent allocation constraints with the time-determining allocation constraints they start and end with.

where the following holds:

- $\forall d \in D : \exists!(d, t_s, t_e) \in \tau$, i.e., each dependent allocation constraint is associated to a pair of time-determining allocation constraints
- $\forall(d, t_s, t_e) \in \tau : t_s < t_e \vee t_s = t_e$, i.e., dependent allocation constraints are associated to consecutive time-determining allocation constraints

The universe of co-allocation constraints is termed CONSTRAINT. □

In the rest of this document, we may occasionally use the simpler term "constraint" if it is clear from the context that we are referring to co-allocation constraints.

We further define the following two functions which return the "first" and "last" time-determining allocation constraint of a co-allocation constraint, according to $<$:

$$\text{firstTac} : \text{CONSTRAINT} \rightarrow \text{TAC}$$
$$cc \mapsto t_f \mid t_f \in cc.T \wedge \forall t \in cc.T, t \neq t_f : t_f < t$$
$$\text{lastTac} : \text{CONSTRAINT} \rightarrow \text{TAC}$$
$$cc \mapsto t_l \mid t_l \in cc.T \wedge \forall t \in cc.T, t \neq t_l : t < t_l$$

Definition 2.29. *Co-allocation Constraint Compliance*
A co-allocation ca is said to **comply with** a co-allocation constraint cc, denoted as $ca \xrightarrow{\checkmark} cc$, if and only if all of the following hold:

- there exists an injective function $\sigma_T : cc.T \rightarrow \wp(\text{COALLOC})$, mapping time-determining allocation constraints to sets of compliant (transitory) allocations within ca: $\forall t \in cc.T : \forall r \in t.R\ \exists! a \in \sigma_T(t) \mid a \in ca \wedge a.r = r \wedge a \xrightarrow{\checkmark} t$
- all transitory allocations compliant with the same time-determining allocation constraint refer to the exact same usage blocks: $\forall t \in cc.T : \forall a_1, \ldots, a_n \in \sigma_T(t) : a_1.U = \ldots = a_n.U$
- the temporal ordering of transitory allocations is consistent with the one defined in the co-allocation constraint: $\forall t_1, t_2 \in cc.T : \forall a_1 \in \sigma_T(t_1), a_2 \in \sigma_T(t_2) : t_1 < t_2 \Rightarrow \text{allocEnd}(a_1) \leqslant \text{allocStart}(a_2)$
- there exists an injective function $\sigma_D : cc.D \rightarrow ca$, mapping dependent allocation constraints to compliant (persistent) allocations: $\forall d \in cc.D : \sigma_D(d) \xrightarrow{\checkmark} d$
- the temporal dependencies between time-determining and dependent allocations are observed: $\forall d \in cc.D, t_1, t_2 \in cc.T : \forall a_1 \in \sigma_T(t_1), a_2 \in \sigma_T(t_2) : \text{allocStart}(a_1) = \text{allocStart}(\sigma_D(d)) \wedge \text{allocEnd}(a_2) = \text{allocEnd}(\sigma_D(d))$

□

The above definition has several consequences. First, the mapping functions are both injective, thus it is permissible to have additional allocations in ca which do not directly satisfy any allocation constraint. In other words: it is allowed to book more resources than actually required (but not less). As a consequence, the mapping functions σ_T and σ_D may not necessarily be unique, i.e., there could be more than one possibility to choose the mappings. Again, this is acceptable as long as there is at least one valid combination.

Definition 2.30. *Co-Allocation Constraint Restriction*
A co-allocation constraint $cc_r = (T_r, <_r, D_r, \tau_r)$ is said to restrict a co-allocation constraint $cc_b = (T_b, <_b, D_b, \tau_b)$, denoted as $cc_r \trianglelefteq cc_b$, if and only if:

- There exists an injective function $\rho_T : T_b \rightarrow T_r$ such that $\forall t \in T_b : \rho_T(t) \stackrel{\cdots}{\trianglelefteq} t$, i.e., each time-dependent allocation constraint in cc_b is mapped to a corresponding augmented restriction in cc_r

- There exists an injective function $\rho_D : D_b \to D_r$ such that $\forall d \in D_b : \rho_D(d) \trianglelefteq d$

and the temporal ordering and dependencies between constraints are preserved:

- $\forall t_1, t_2 \in T_b : t_1 <_b t_2 \Rightarrow \rho_T(t_1) <_r \rho_T(t_2)$
- $\forall (d, t_1, t_2) \in \tau_b \; \exists (\rho_D(d), \rho_T(t_1), \rho_T(t_2)) \in \tau_r$

□

The implications of co-allocation constraints restriction are analogous to the restrictions of the individual allocation constraints: any co-allocation compliant with cc_r is compliant with cc_b, but not every co-allocation compliant with cc_b is necessarily compliant with cc_r.

While it is somewhat of an anticipation of the contents of the subsequent sections, the following example, based on Figure 2.6 (page 19) may be helpful to better understand (co-)allocation restrictions. Examining its structure, the top row of the figure corresponds to a sequence of time-determining allocation constraints, namely for incoming bandwidth, execution (CPU), and outgoing bandwidth. This, along with the dependent allocation constraints, is information obtained from the provider of the operation. Note that (in this row) data input and output concern only the host providing the co-allocation. The second row, presenting the associated data output and input (at the other endpoint), is dynamic in the sense that it depends on which workflow engine is actually invoking the operation, and is added as augmented restrictions to the basic (top-row) time-determining allocation constraints. By definition, the allocation constraints mandate the same usage blocks for both the incoming and outgoing resource.

Co-Allocation Constraint Graphs

A co-allocation constraint allows to express the fact that multiple resources need to be used in a coordinated manner in order to perform one logical operation. An important property of co-allocation constraints is that all contained time-determining co-allocation constraints must be totally ordered, i.e., be sequential. In analogy to workflows, which are graphs of operations, one can also define graphs of co-allocation constraints (in fact, as we shall see in Sections 2.6 and 2.7, DWARFS uses such graphs of co-allocation constraints to reason on workflows).

Definition 2.31. *Co-Allocation Constraint Graph*
A co-allocation constraint graph cg represents a multiset of co-allocation constraints and their temporal dependencies. It is defined as a strict partial order $cg = (CC, <)$, where all of the following hold:

- CC is a non-empty multiset of co-allocation constraints
- $<$ is a non-reflexive binary relation on CC, indicating the temporal relationship between the co-allocation constraints
- There exists exactly one co-allocation constraint which precedes all other constraints: $\exists! s \in CC : \forall cc \in CC \mid cc \neq s : s < cc$

- There exists exactly one co-allocation constraint which follows all other constraints: $\exists! e \in CC : \forall cc \in CC \mid cc \neq e : cc < e$

The universe of co-allocation constraint graphs is termed CONSTRAINTGRAPH. □

In the context of this work, co-allocation constraint graphs are used as another means to represent and reason on workflows. Just as workflows must have single start and end activities, co-allocation constraint graphs must have one "first" and one "last" constraint, which is made explicit by the latter two conditions in the definition. We define the following two operations on co-allocation constraint graphs, which return those constraints:

$$\begin{aligned} \text{firstConstr} : \text{CONSTRAINTGRAPH} &\rightarrow \text{CONSTRAINT} \\ cg &\mapsto s \\ \text{lastConstr} : \text{CONSTRAINTGRAPH} &\rightarrow \text{CONSTRAINT} \\ cg &\mapsto e \end{aligned}$$

Note that it is of course possible to define a co-allocation constraint graph cg_s consisting of a single co-allocation constraint $cc_s : cg_s = (\{cc_s\}, \emptyset)$.

2.5 Reservations and Infrastructure State

So far, we have introduced the basic concepts that allow us to define resources and their usages, but have not yet taken the step to actually employing these concepts within the context of Advance Reservations.

2.5.1 Reservations

Intuitively, a reservation is simply an acknowledged set of allocations, where *acknowledged* means that the providers of the respective resources have committed to holding these resources back and making them available at the time needed.

Definition 2.32. *Reservation*
A reservation rs is a tuple $rs = (ca, t)$, where:

- $ca \in$ COALLOC is a co-allocation
- $t \in \mathbb{N}$ is the timestamp at which the reservation was acknowledged

The universe of reservations is termed RESERV. □

Definition 2.33. *Reservation Cost*
The cost of a reservation is determined by the following function:

$$\begin{aligned}\text{reservCost} : \textsc{Reserv} &\to \mathbb{R}_+ \\ (ca, t) &\mapsto \text{coallocCost}(ca, t)\end{aligned}$$

□

There are two noteworthy aspects regarding reservations: First, they concern entire co-allocations instead of the more basic allocations. Second, they bear a timestamp at which they were "created". Reserving co-allocations instead of the individual allocations makes sense because by definition, these allocations are interdependent and, put bluntly, only "useful" in combination. The timestamp keeps track of when a reservation was created, but more importantly, it also fixes the reservation cost. This second property could not be achieved if reservations concerned individual allocations, as the cost of a co-allocation is not necessarily equal to the sum of the cost of its allocations.

2.5.2 State and State Changes

Reservations have one intrinsic property, namely that after they are made, they occupy the resources reserved, which means that they may hamper the feasibility of future allocations. In other words, we are dealing with a system which is changing its state as reservations are made (or cancelled).

Definition 2.34. *Reservation State*
A reservation state S is a multiset of reservations: $S \subseteq \tilde{\wp}(\textsc{Reserv})$.[1] The universe of reservation states is termed $\textsc{State}$.

□

Just like individual allocations may be valid or invalid (if they surpass their resource's capability at any point in time), we can define the validity of an entire reservation state in a similar way: informally, a reservation state is invalid if the sum of the reservations would overbook any contained resource at any time.

Definition 2.35. *Reservation State Validity*
A reservation state S is valid, if and only if

$$\nexists r \in \textsc{Resource} \mid \sum_{(u_i, r) \in \bigcup\limits_{(ca,t) \in S} ca} (u_i, r) \text{ is not valid}$$

□

[1]The symbol $\tilde{\wp}$ refers to the power multiset, as defined in [SIY08]

State changes

Introducing new reservations, or cancelling existing ones, corresponds to a transition from one reservation state to another. We define two functions, *reserve* and *cancel*, to induce such state changes. The *reserve* operation, given a reservation state, a timestamp, and a multiset of co-allocations, returns a new reservation state which includes reservations for the requested co-allocations. Note that if there are timing inconsistencies, or if some requested allocations cannot be accommodated in addition to reservations that are already in place, no new state is produced (i.e., the state is returned unchanged). Cancelling existing reservations is similar, but slightly simpler.

Definition 2.36. *Co-Allocation Feasibility*
The feasibility of a multiset of co-allocations is defined as the possibility to commit (or reserve) all of the contained co-allocations simultaneously at a given timestamp, in a certain reservation state. Feasibility is defined through the function *feasible*:

$$\text{feasible} : \textsc{State} \times \mathbb{N} \times \tilde{\wp}(\textsc{CoAlloc}) \to \mathbb{B}$$
$$(S, t, CA) \mapsto \begin{cases} \texttt{false} & \text{if } t \leqslant \max\limits_{(ca_e, t_e) \in S} t_e \vee \\ & S \cup \{(ca, t) \mid ca \in CA\} \text{ is not valid} \\ \texttt{true} & \text{otherwise} \end{cases}$$

□

Definition 2.37. *Co-Allocation Commitment*
Simultaneous commitment (or reservation) of a multiset of co-allocations is defined through the function *reserve*:

$$\text{reserve} : \textsc{State} \times \mathbb{N} \times \tilde{\wp}(\textsc{CoAlloc}) \to \textsc{State}$$
$$(S, t, CA) \mapsto \begin{cases} S \cup \{(ca, t) \mid ca \in CA\} & \text{if feasible}(S, t, CA) \\ S & \text{otherwise} \end{cases}$$

□

Definition 2.38. *Reservation Cancellation*
Simultaneous cancellation of a multiset of co-allocations is defined through the function *cancel*:

$$\text{cancel} : \textsc{State} \times \tilde{\wp}(\textsc{Reserv}) \to \textsc{State}$$
$$(S, R) \mapsto S \setminus R$$

□

The notion of time

As timestamps are used quite extensively in the previous definitions, it may be useful to recapitulate a few aspects here. First, we assume that time is continuously moving forward, and that the system is in exactly one reservation state at any given time. Second, the timestamps at which reservations are *made* simply correspond to the timestamp at which the system becomes *aware* of the contained co-allocations (and therefore allocations), but these are independent of the start and end timestamps of the contained allocations. As the term *Advance Reservation* implies, the reservation timestamp will normally predate the allocations' timestamps. However this is not a strict requirement, especially in settings where it may be allowed to revise (for example, extend) currently active reservations (active reservations at a timestamp t are simply those whose allocations encompass t).

2.6 Workflow Execution in DWARFS

The preceding sections introduced, among others, operations, workflows, resources, and reservations. There are still a few pieces missing before the full picture of DWARFS is complete. Before we engage in further definitions, it is helpful to see how a workflow execution in DWARFS takes place.

2.6.1 Workflow Orchestration and Physical Data Flow

Section 2.2.3 presented the logical structure of a workflow (more precisely: a workflow description, or definition).[2] While such workflow descriptions are powerful means to *define* new functionality, they must also be leveraged – in other words, executed – to actually *provide* that functionality.

DWARFS is explicitly geared at Service-Oriented Architectures, where the provision of specific operations is not limited to a particular entity. Rather, multiple, possibly competing, organizations may provide functionally equivalent services. Thus, one operation could be available at several independent *Operation Providers* which may well be distributed globally.

But the same holds for the execution of workflows themselves: The ability to execute and combine the steps of a particular workflow, given its description, is not limited to a single entity, but can again be offered by a number of independent providers, or *Workflow Engines*. Moreover, since a workflow definition usually contains more than a single activity, workflow execution can also be accomplished as a *coordinated* effort of multiple workflow engines, thus executing the workflow itself in a distributed manner.

[2] We will mostly use the terms "workflow definition" and "workflow description" interchangeably in the text, except when in formal definitions. The term "workflow" may also sometimes be used loosely to refer to other aspects, such as "workflow execution", but it should always be intuitively clear what exactly the wording refers to in the context.

Consider again Figure 2.2, which shows the sample workflow description and its logical data flow. Figure 2.8 depicts how such a workflow is actually executed in DWARFS. There are several things to note.

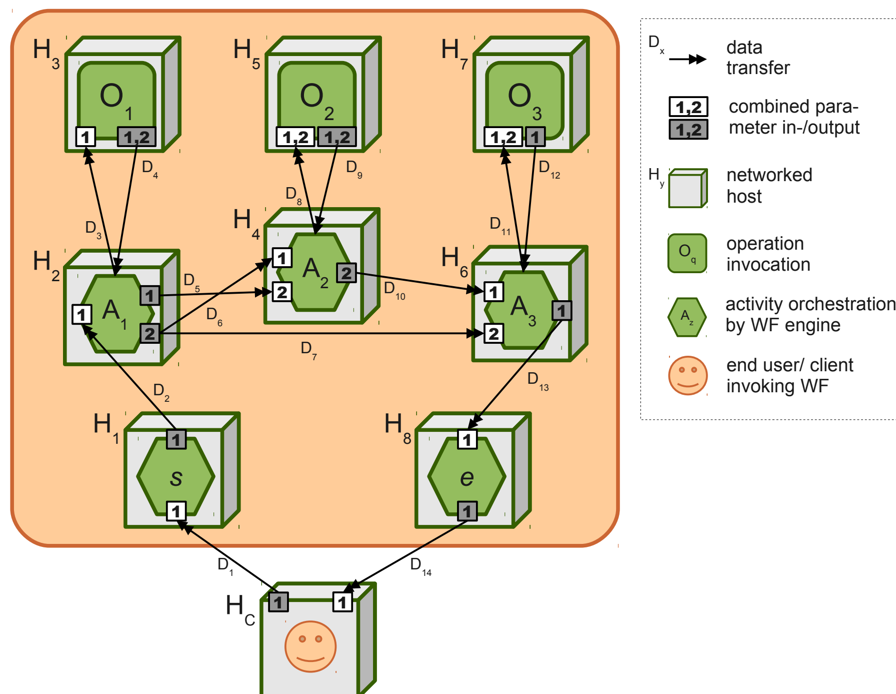


Figure 2.8: Sample Workflow, with Physical Data Flow

First, each activity of the workflow is orchestrated by one Workflow engine, which takes care of handling the inputs and outputs according to the workflow definition. The actual invocation of an operation at a suitable provider (and possibly on a different host) is then handled by the responsible workflow engine.

Second, during the orchestration, workflow engines may receive multiple input parameters from different input sources, and forward output parameters to multiple destinations. However, this does not hold for the operation providers: an operation invocation is (logically) a single operation following the request-response pattern. While responses may be delivered asynchronously if the processing takes a long time, an operation still takes one set of inputs (delivered in a single transfer), and produces one set of outputs (delivered in a single transfer). For example, the input parameters for O_3 are gathered using two separate transfers D_7 and D_{10} by the workflow engine orchestrating A_3, but are sent in a single transfer for O_3 to be executed (D_{11}).

Third, and along with the previous observation, operation outputs are always produced in full by operation providers. Even if part of the output is not further used within the workflow, the operation provider still produces the full result, which means that the invoking workflow engine will still receive it, but can then silently discard it. An example is the invocation of O_2, where only the second output is used in the workflow.

The Resource Usage and Timing Perspective

Figure 2.8 depicts in detail *how* a workflow execution is performed, and gives a more fine-grained view on how data transfers and operation invocations relate to each other (in terms of dependencies, and therefore temporally). However, using only that information, it is impossible to determine exactly how long an operation invocation or a data transfer takes – quite simply because in general, one does not know the exact data that is being used.

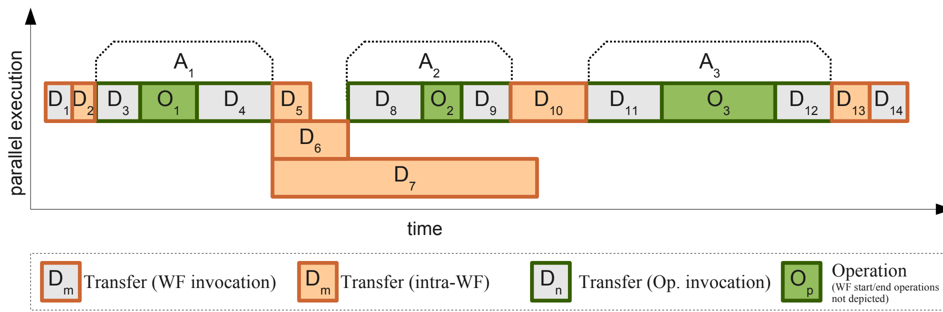


Figure 2.9: Sample Workflow – Timing

Let us for a moment ignore this fact and take a look at a workflow execution that has already taken place (in any system with, or without advance reservations). Figure 2.9 shows an example. While the actual timings in the example are arbitrary, the *structure* of the result is the one any real execution will yield, and all delays are effectively introduced by capacity limitations of some resource.

In fact, if we take such a ”post-mortem“ look at a workflow execution which has taken place, we can say that retrospectively, we are always able to determine the reservations which would have been required for its execution. Simply put, our goal is to also do this prospectively, for executions which have not yet taken place.

If we take a closer look at Figure 2.9 and Figure 2.6, we notice that they are very similar in appearance. This is not a coincidence, because just as a workflow definition is a combination of operations, we can regard a workflow execution as a combination of co-allocations corresponding to the operation invocations. And, taking this one step further, a workflow description can then simply be regarded as a combination of co-allocation constraints, which are derived from the individual operations.

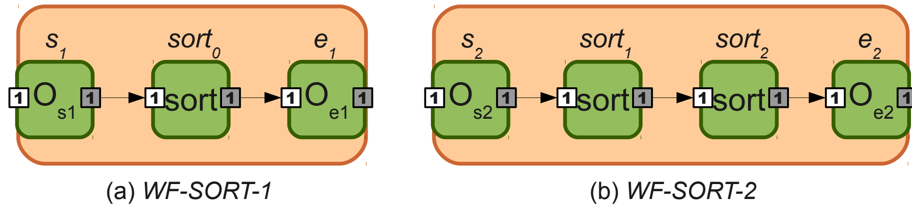
In this section, we will provide the final definitions which bring together the previously defined concepts of operation, workflow, and resource, so that we end up with a complete set of definitions allowing to describe an entire DWARFS infrastructure.

2.6.2 Predicting Workflow Executions

As previously described, once a workflow execution has taken place, we are able to (retrospectively) observe all of the involved data, resources, and temporal behavior of the execution. In fact, these are all related: the temporal behavior is determined by the resources that have been used in order to perform operations on data. There are two evident, yet crucial observations to be made: first, the data produced by a workflow's

operations is not known before the execution takes place. This is obviously true for the final output of the workflow, as well as intermediate data which is only used internally during its execution. Stating it differently: if the output had already been known before executing a workflow, there would normally be no point in executing the workflow at all. Second, we assume that operations behave in a deterministic manner, so both the output of an operation invocation, as well as its runtime behavior, depend largely on the input data. Note that these observations, again, represent the most general case: Of course there may be the need to run a scientific workflow even if the output is known beforehand – for example to verify previous results, or if it is some side effects, rather than the output, that one is interested in; similarly, not every operation's runtime behavior necessarily changes with the input. However, these are simply special cases that are also encompassed by the subsequent statements.

To ease the following explanations, we consider as running examples two trivial workflows, which both allow to sort a list of numbers (of arbitrary length). Figure 2.10 depicts two such workflows. In particular, the second workflow contains a second sort operation, which is redundant and useless (but does not affect the correctness of the result); this was done on purpose, and will subsequently be helpful to illustrate some of the concepts.



(a) *WF-SORT-1* (b) *WF-SORT-2*

Figure 2.10: Workflow Definitions: Sorting Lists of Numbers

According to the definitions in Section 2.2, we can view these workflows as operations: *WF-SORT-1* $:= (I_1, O_1, \phi_1)$ and *WF-SORT-2* $:= (I_2, O_2, \phi_2)$, with $I_1 = (\mathbb{N}^p), O_1 = (\mathbb{N}^p), I_2 = (\mathbb{N}^q), O_2 = (\mathbb{N}^q)$. The *sort* operation used internally has a similar signature: $sort := (I_s, O_s, \phi_s)$, with $I_s = (\mathbb{N}^r), O_s = (\mathbb{N}^r)$. Note that all the indicated cardinalities are not fixed, but "dynamic", meaning that one can sort inputs of arbitrary sizes. For completeness, we also give the formal definition of ϕ_1 (the other two functions are analogous):

$$\begin{aligned}\phi_1 : \mathbb{N}^p &\rightarrow \mathbb{N}^p \\ (i_1, \ldots, i_p) &\mapsto (o_1, \ldots, o_p) \mid o_x \leqslant o_y \Rightarrow x \leqslant y \\ &\qquad \wedge \forall a \in [1, p]\ \exists b \in [1, p] : i_a = o_b\end{aligned}$$

We will now examine some characteristics of (potential) workflow executions, where we know some details about the workflow's structure and the constituting operations' implementation, as well as some characteristics of the input (but not the exact input). Let us assume that we have a list of numbers l, which shall be sorted using *WF-SORT-1* or *WF-SORT-2*. The output produced by both workflows will be identical, but their runtime behavior will differ. Let n denote the length of the input, i.e., $n := |l|$.

Here are some general statements about various well-known implementations of the sort operations:

- Merge sort has a worst-case asymptotic runtime of $O(n\ log\ n)$, requiring $\theta(2n)$ space.
- Bubble sort runs in $O(n^2)$, requiring $\theta(n)$ space.
- Heapsort runs in $O(n\ log\ n)$, requiring $\theta(n)$ space.
- Evidently, the output of all implementations is sorted.
- If the input is already sorted, merge sort and bubble sort runtime is $O(n)$, while heapsort's runtime remains $O(n\ log\ n)$.

The above statements are using the asymptotic notation and therefore "abstract away" the concrete factors. However, knowing the exact implementation and the physical capabilities of the machine providing the operation, it is possible to provide more concrete bounds, for instance to replace $O(n\ log\ n)$ by $k * n\ log\ n + p$, where k and p are instance-specific factors allowing to calculate an actual upper bound of wall-clock runtime.

Let us now consider the data that is processed. While we do not know the exact data involved, we do know that the input is a list of n numbers, and the output is a *sorted* list of n numbers. Assuming a very simple encoding, using a binary 64-bit representation of the numbers (with no other overhead), the physical transfer size of any list of n numbers would be exactly $8n$ bytes. When using a more sophisticated encoding such as SOAP, there will be additional overhead required for the protocol, but it is still possible to determine upper bounds for the message size.

While we will further discuss this soon, the idea is to describe and classify the input data with sufficient detail – but without needing to know the actual data –, so that one can predict both the size of the data to be processed, as well as the behavior of the involved operations. In other words, we are performing calculations and predictions based on metadata. Most importantly, such metadata is available not only for the (prospected) workflow input, but also transitively derived for the data flow inside the workflow.

For a concrete example, consider *WF-SORT-2*, assuming that both sort activities use bubble sort and take place on the same slow machine, which has a worst-case wall-clock runtime of $42 * n^2$ microseconds for unsorted input, and $42 * n$ microseconds for pre-sorted input. For a workflow input of $n = 1000$ numbers, the $sort_1$ invocation could be determined to take (at most) 42 seconds, while the invocation of $sort_2$ would take .0042 seconds. One would also know that all data transfers, using the abovementioned encoding, would be of size 8 kB.

The following sections go into more detail and formalize the concepts we have just introduced.

2.6.3 Data Characteristics

Data characteristics represent metadata about data. In a sense, they are somewhat similar to data types as introduced in Section 2.2, but provide additional information about the concrete instance of data: while data types represent (only) a domain, data characteristics allow for a finer-grained classification of data. Furthermore, they allow to determine (an upper bound of) the physical size of the representation of the data, and may also convey semantic information. In order to keep the definition simple and concise, but flexible and extensible enough to accommodate for many kinds of information, we define data characteristics as a simple combination of data attributes.

Definition 2.39. *Data Attribute*
A data attribute provides information about a single aspect of a data instance. The universe of data attributes is termed DATTR. □

Definition 2.40. *Data Characteristic*
A data characteristic dc is a set of data attributes, $dc = \{da_1, \ldots, da_n\}, da_i \in$ DATTR. The universe of data characteristics is termed DCHAR. □

As outlined previously, data characteristics (and therefore data attributes) are meant to describe the properties of data, but without necessarily knowing their exact values. The goal is to reason about an expected (future) execution and data flow of a workflow, without actually having executed the workflow yet. To come back to the previous example, a sort operation could characterize its output as {SORTED, LIST1000} given an input characterization of {UNSORTED, LIST1000}.[3] If the input characteristics are {SORTED, LIST1000}, the output characteristics will not change, but the prospected runtime for invoking the operation may.

Data Characteristics-related Functions

Data characteristics provide information (or metadata) about the data that is consumed and produced during a workflow execution. Such metadata is important information for the individual activities of the workflow (i.e., the operations contained therein), which need to "understand" the semantics of the characteristics. However, from the point of view of workflow execution itself, such a deep understanding is not a necessity: to put it bluntly, actual workflow orchestration is merely a coordinated "data shipping" between operation providers, where the content of the data does not matter, and can be treated as a black box without the need to understand it. There is one exception however: because we are predicting the runtime behavior of the workflow, we do need to know the *size* of the (expected) data, as this in turn affects the timing of the network transmissions.

On the other hand, just as a workflow definition needs to preserve type compatibility (cf. Section 2.2.3), not every data characteristic is applicable for every data type: for in-

[3]The data attributes used are merely examples that could apply to characterize lists of numbers. The semantics of the names used here should be self-explaining.

stance, the data attribute SORTED will likely not make sense for a data type representing a (single) image.

Thus, we assume the existence of the following two functions related to data characteristics:

$$\text{size} : \text{DCHAR} \to \mathbb{N}_+$$
$$\text{characterizes} : \text{DCHAR} \times \text{TYPE} \to \mathbb{B}$$

2.6.4 Endpoints, Operation Instances, and Workflow Engines

The previous sections have already mentioned the fact that runtime behavior of an operation invocation for some input depends not so much on the operation itself (*what*), but mostly on the actual implementation (*how*), as well as the resources of the machine(s) executing it (*where*).

Endpoints

In Section 2.4, we introduced resource providers as the general term for an entity managing multiple related, usually physically co-located, resources. An endpoint is simply a resource provider which is reachable via network, i.e., which contains (at least) two resources of resource type BANDWIDTH, denoting its incoming and outgoing bandwidth respectively. Note that not every resource provider is necessarily an endpoint. In particular, one may want to consider a network link itself (i.e., the infrastructure as provided by an ISP, connecting physical endpoints and possibly limiting the capacity of their connectivity) as an abstract resource which can be considered during network transfers and be managed accordingly.

Definition 2.41. *Endpoint*
The universe of endpoints is a subset of the universe of resource providers, and termed ENDPOINT: ENDPOINT $\subseteq$ RESPROV.

We define the following functions for endpoints:

$$\text{incoming} : \text{ENDPOINT} \to \text{RESOURCE}$$
$$ep \mapsto r_i : r_i \in ep \wedge r_i.t = \text{BANDWIDTH} \wedge r_i \text{ is } ep\text{'s incoming bandwidth resource}$$

$$\text{outgoing} : \text{ENDPOINT} \to \text{RESOURCE}$$
$$ep \mapsto r_o : r_o \in ep \wedge r_o.t = \text{BANDWIDTH} \wedge r_o \text{ is } ep\text{'s outgoing bandwidth resource}$$

□

Note that while one could associate the notion of endpoint with an individual computer, or host, the term is deliberately more general, because it may encompass entire clusters of computers, load-balanced machines etc. As long as resource management is correctly provided, any such setup can qualify as an endpoint.

The following two entities (workflow engines and operation instances) both provide their functionality over the network, and are thus always associated with an endpoint.

Workflow Engines

Definition 2.42. *Workflow Engine*
A workflow engine is an entity capable of orchestrating workflow execution. The universe of workflow engines is termed ENGINE.

The endpoint associated with a workflow engine is determined by the following function:

$$\text{engEndp} : \text{ENGINE} \rightarrow \text{ENDPOINT}$$

□

Operation Instances

An operation instance represents the deployment of an operation at an endpoint. While it is clear that such a deployment necessarily also involves a particular (algorithmic) implementation of the operation, it is not necessary to represent the actual implementation in the formal model. On the other hand, there may well be multiple instances of the same operation deployed on the same endpoint.

Definition 2.43. *Operation Instance*
An operation instance oi is a tuple $oi = (op, ep, id)$, where:

- $op \in \text{OP}$ is the operation that the instance provides.
- $ep \in \text{ENDPOINT}$ is the endpoint where the operation instance is deployed.
- id is a unique identifier representing the operation instance.

The universe of operation instances is termed OPINST. □

Note that the *id* element is globally unique, i.e., $\forall oi_1, oi_2 \in \text{OPINST} : oi_1.id = oi_2.id \Rightarrow oi_1 = oi_2$. There are two reasons for the existence of this element: from a formal perspective, it allows to express the deployment of multiple instances of the same operation on the same endpoint. When considering the implementation of the model, it could contain the physical URL where the implementation can be reached (for instance its Endpoint Reference, when considering a SOAP-based implementation).

Finally, workflows are themselves operations. Thus, *every* workflow engine that knows, or can be made to know, about a particular workflow definition wd provides corresponding operation instances; an exact explanation and definition is given in Section 2.7.4.

2.6.5 Data Transfers

Data transfers always take place between endpoints, originating at an outgoing bandwidth resource and terminating at an incoming bandwidth resource. In the simplest case, a network transfer between endpoints ep_{out} and ep_{in} thus affects the set of resources $\{\text{outgoing}(ep_{out}), \text{incoming}(ep_{in})\}$. If more information about the network connectivity and infrastructure is available (i.e., if affected networks, or concrete "hops", between ep_{out} and ep_{in}, can be determined and are represented as resources), the set is extended with these participating resources . We thus assume the existence of a function $transferRes$ which can determine the set of resources affected by such a data transfer:

$$\begin{aligned} \text{transferRes} : \textsc{Endpoint} \times \textsc{Endpoint} &\to \wp(\textsc{Resource}) \\ (ep_{out}, ep_{in}) &\mapsto \{r : r.t = \textsc{Bandwidth} \wedge r \text{ is affected} \\ &\qquad \text{by network transfers from } ep_{out} \text{ to } ep_{in}\} \end{aligned}$$

Because all affected bandwidth resources need to be co-allocated for such a network transfer, we can define a function $transferTac$ which returns a time-determining allocation constraint for network transfers of a given size:

$$\begin{aligned} \text{transferTac} : \textsc{Endpoint} \times \textsc{Endpoint} \times \mathbb{N}_+ &\to \textsc{Tac} \\ (ep_{out}, ep_{in}, size) &\mapsto (\text{transferRes}(ep_{out}, ep_{in}), 1, \\ &\qquad \min_{r \in \text{transferRes}(ep_{out}, ep_{in})} (r.c), size) \end{aligned}$$

The time-determining allocation constraint's sum is the size to be transferred, while the maximum resource usage is determined by the affected resource with the lowest capacity, and the minimum is the minimal allowed value 1.

2.6.6 Operation Invocation

The previous definitions allow to formalize the following question: "when invoking a particular operation instance with parameters characterized by particular data characteristics, what will the characteristics of its output be, and which resources will be required for its execution?"; from an implementation point of view, this is a meta-operation that must be provided by the operation instance itself, and of course the result depends on all of the input data characteristics, the (algorithmic) implementation of the operation, and the resources of the particular endpoint (i.e., hardware limitations). Formally, we view this as a global operation, which is defined below.

Definition 2.44. *Invocation Characteristic*
An invocation characteristic ic comprises information about both the invocation's output, as well as the resources required during the invocation. It is defined as a tuple $ic = (d, cg)$, where:

- $d \in \textsc{DChar}^n$ is a tuple of n data characteristics

- $cg \in$ CONSTRAINTGRAPH is a co-allocation constraint graph

The universe of invocation characteristics is termed INVCHAR. □

Invocation characteristics are the result of the invocation of a particular operation instance with particular input data, where that invocation is performed (i.e., called) by a particular remote endpoint. For reasons of clarity, we have split up this determination of invocation characteristics into multiple functions, as described below.

Because operation instances are supposed to operate in a deterministic fashion, the output data characteristics of an invocation depend solely on the input data characteristics, and are independent of the endpoint performing the invocation. However, this does not hold for the co-allocation constraint graph: in particular, the co-allocation constraints representing the data upload (input) and download (output) *do* depend on the particular endpoint which performs the invocation. This can be conceptually represented as two closely related functions *execChar* and *invokeChar*, which are described below.

Determination of Invocation Characteristics (Execution only)

Let *execChar* denote a function which determines invocation characteristics for a particular operation instance, given the data characteristics of the input data, but without considering the endpoint performing the invocation:

$$\text{execChar} : \text{OPINST} \times \text{DCHAR}^n \rightarrow \text{INVCHAR}$$

Note that the data characteristics for input and output must match the operation's signature, i.e., the cardinalities are correct, and the characteristics are applicable for the respective data types. More formally, if $((dc_1^{out}, \ldots, dc_m^{out}), cg) = \text{execChar}(oi, (dc_1^{in}, \ldots, dc_n^{in}))$, then both of the following must hold:

- $|oi.op.I| = n \wedge \forall p \in [1, n] : \text{characterizes}(dc_p^{in}, oi.op.I[p]) = \texttt{true}$
- $|oi.op.O| = m \wedge \forall q \in [1, m] : \text{characterizes}(dc_q^{out}, oi.op.O[q]) = \texttt{true}$

We also assume that the output is as concise as possible with regard to the data characteristics, in order to convey as much information as possible: For instance, if an operation instance guarantees that the output will be sorted, it should be characterized as such; if an instance "knows" that its output is always empty for a particular input class, it should be characterized as such, etc.

The above definition allows to determine co-allocation constraints, and the data characteristics of the operation output, for a particular operation instance. An important observation is that even for two operation instances referring to (i.e., implementing) the same operation, the returned co-allocation constraints will most likely differ, because of hardware or implementation differences. The same might even be true for the data

characteristics of the output – for instance, different operation instances may produce output optimized in different ways, which can be reflected in the data characteristics.

The co-allocation constraint graph cg reflects the resources required for the invocation. Because we are operating in an SOA setting where invocations are performed over the network, cg must contain constraints for the networking resources relating to data input and output as the first and last constraint, respectively. However, as the invoking endpoint is not known at this point, these constraints are necessarily incomplete in the sense that only one end of the network transfers is considered, namely the endpoint of the operation instance itself. More specifically, and in terms of resources, the following holds:

$$\text{firstTac}(\text{firstConstr}(cg)).R = \{\text{incoming}(oi.ep)\}$$
$$\text{lastTac}(\text{lastConstr}(cg)).R = \{\text{outgoing}(oi.ep)\}$$

For a graphical depiction, refer to Figure 2.6 (page 19); these co-allocation constraints correspond to the leftmost and rightmost co-allocations in the topmost row of the figure.

Determination of Invocation Characteristics (Remote Invocation)

Let *invokeChar* denote a function which determines the invocation characteristics of a particular operation instance, given the data characteristics of the input data, and the concrete endpoint performing the operation. Put simply, the output of *invokeChar* depends on the output of *execChar*, but contains a more "complete" co-allocation constraint graph which considers all required network resources.

$$\text{invokeChar} : \textsc{OpInst} \times \textsc{Endpoint} \times \textsc{DChar}^n \to \textsc{InvChar}$$
$$(oi, ep, dc) \mapsto \text{augment}(\text{execChar}(\text{oi}, \text{dc}), ep)$$

2.7 Workflow Schedules

In order to provide a predictable execution of workflows, DWARFS uses Advance Reservations of the resources required for its execution. This implies that such reservations are set up before the execution can be started – in other words, a workflow adheres to a previously established workflow schedule.

More concretely, a workflow schedule associates a workflow description with physical endpoints that are used during the execution, and with a number of co-allocations, such that all co-allocation constraints resulting from the physical associations are fulfilled.

Definition 2.45. *Workflow Schedule*
A workflow schedule ws is a tuple $ws = (wd, u, D, \Xi, \Omega, C)$, where:

- $wd \in \textsc{Wfd}$ is a workflow description
- $u \in \textsc{Endpoint}$ is the endpoint invoking the workflow

- $D \in \text{DCHAR}^n$ is a tuple containing the data characteristics of the input to the workflow
- Ξ is a mapping function associating each of wd's activities with a workflow engine: $\Xi : wd.A \rightarrow \text{ENGINE}$
- Ω is a mapping function associating each of wd's activities, except for its start and end activity, to a correct operation instance: $\Omega : wd.A \setminus \{wd.s, wd.e\} \rightarrow \text{OPINST}$, where $\forall a \in wd.A \setminus \{wd.s, wd.e\} : \Omega(a).op = wd.\omega(a)$
- $C \in \tilde{\wp}(\text{COALLOC})$ is a multiset of co-allocations, encompassing all required co-allocations for the execution.

  The universe of workflow schedules is termed WFS.

□

There are many further conditions that need to be fulfilled for a workflow schedule to actually be *valid*, which will be covered later in this chapter. However, before we go into detail about these conditions, it is helpful to take a closer look at how such a schedule may actually be determined.

2.7.1 Determining Workflow Schedules

Looking at the definition of a workflow schedule, one could consider it as having both "static" and "dynamic" elements: wd, u, and D are static in the sense that they represent exactly one value which does not change. On the other hand, Ξ, Ω and C are variable in the sense that there are multiple possibilities to assign these values. In other words, if one looks at a workflow schedule as the solution to a scheduling problem, then the former elements represent the *input* of the problem, while the latter represent its *output*. In the following, we shortly describe, from a logical perspective, how these values could be determined in an incremental manner.

Operation Instance Selection

We start by selecting an operation instance for each activity, i.e., by defining the mapping function Ω.

Let *instCount(o)* denote the number of instances for a particular operation $o \in \text{OP}$: $\text{instCount}(o) := |\{oi \in \text{OPINST} : oi.o = o\}|$. For a workflow with n operations $o_1, \ldots, o_n$ (excluding s and e), there are $\prod_{i=1}^{n} \text{instCount}(o_i)$ possibilities of choosing operation instances.

Figure 2.11 shows the same workflow as Figure 2.8, but again from a different perspective: In this figure, the characteristics of the workflow input (C_1) are known, and for each of the contained operations, one particular instance has been selected and fixed. The edges where data transfers are performed have been annotated with the

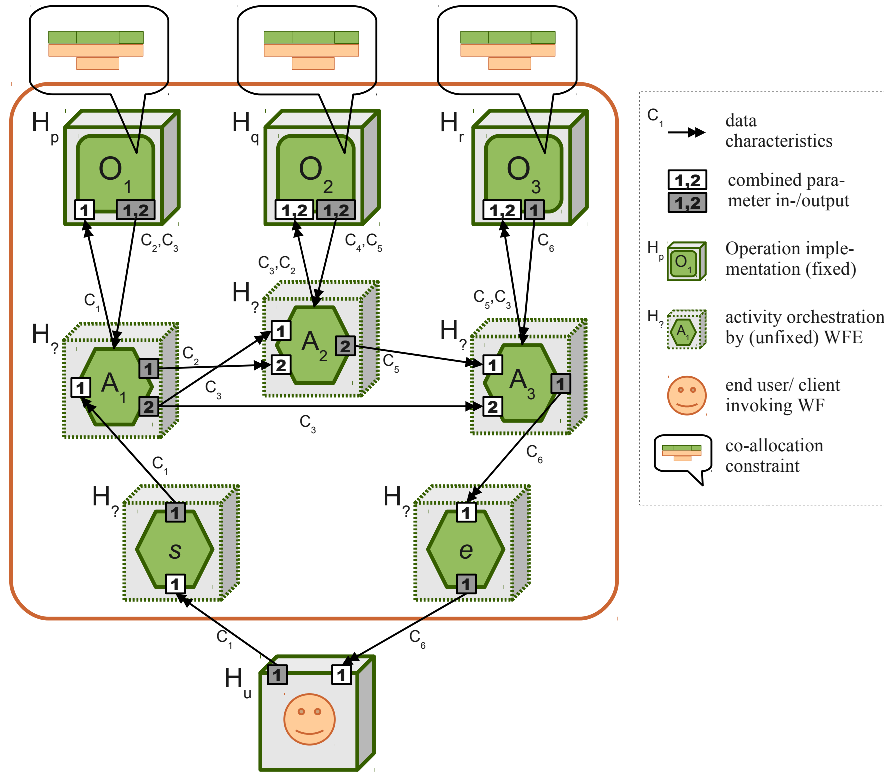


Figure 2.11: Workflow Scheduling: Operation Instances

corresponding data characteristics, and the operation instances with the respective co-allocation constraints. All of these metadata are transitively determined from the chosen operation instances and previously provided or derived metadata: C_2 and C_3 are determined by $invokeChar(O_1, (C_1))$, and so on.

At this stage, contrary to the operation instances, the workflow engines handling the individual activities have not been fixed yet. In other words, we do not yet know all of the endpoints for the data flow. However, we already do know the size of each data transfer, as it can be derived from the data characteristics via the *size* function.

Workflow Engine Selection and Inter-Engine Co-Allocation Constraints

The second task of scheduling involves the fixation of the workflow engines involved in the execution of the workflow, i.e., fixing Ξ. For a workflow with n activities, and in an infrastructure with m available workflow engines, this results in $(n-1)^m$ possible assignments (the start and end activities must be co-located on the same engine). Once the assignments are fixed, the co-allocation constraints of the individual operation instances, or more precisely, the time-determining allocation constraints which refer to bandwidth therein (data upload and download) are augmented with the respective bandwidth resources of the invoking engine's endpoint. Since the endpoints of the inter-workflow engine transfers (i.e., the transfers represented by $wd.\delta$), and for the input and output of the workflow itself, are known as well, new co-allocation constraints can then be created for these transfers.

The result is depicted in Figure 2.12, where all activities and data transfers are encompassed by co-allocation constraints.

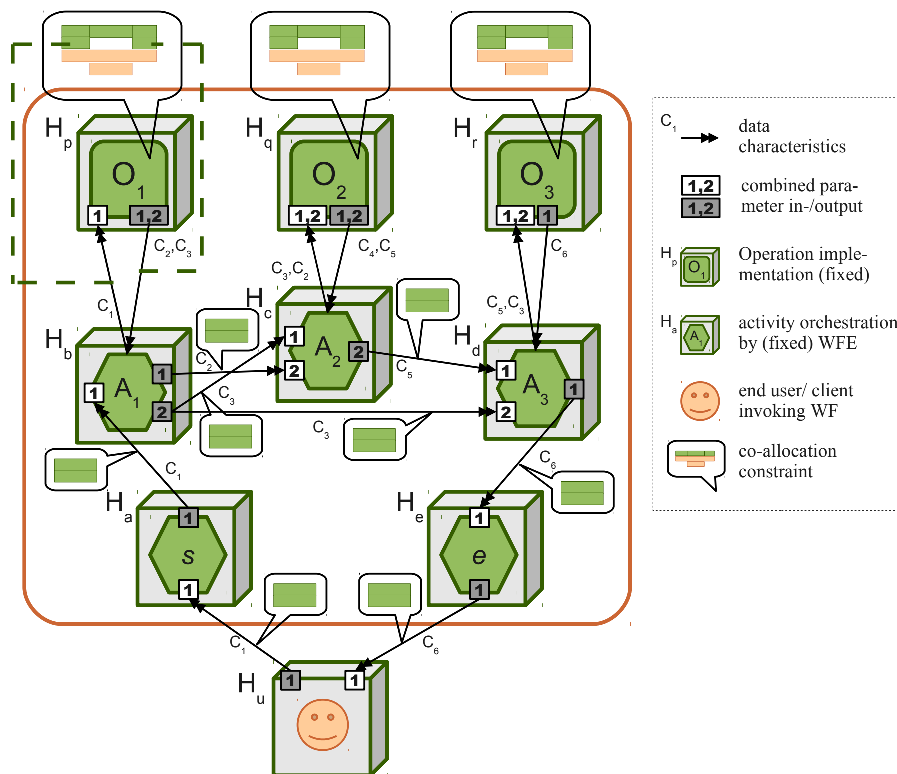


Figure 2.12: Workflow Scheduling: Workflow Engines and Inter-Engine Constraints

Determination of Co-allocations

The final element in a workflow schedule, ie the multiset of co-allocations C, may be determined by finding co-allocations such that all constraints imposed by Ξ and Ω are satisfied, and the individual co-allocations respect the workflow's dependencies.

2.7.2 Functions relating to Workflow Schedules

In the following, we introduce a number of functions, geared at particular aspects of a workflow schedule. These functions are meant to split the subsequent definitions into more manageable, and easily understandable blocks. Note that all of these definitions are meant to be interpreted in the context of a workflow schedule – in other words, the entities $ws, wd, u, D, \Xi, \Omega, C$ are "predefined" according to the definition of a workflow schedule.

Data Characteristics

Let *inChar* and *outChar* denote functions which determine the data characteristics of the input and output of some activity of the workflow definition *wd*. As stated earlier, most

of this metadata is transitively determined through previous metadata and the chosen operation instance mapping. Therefore, these operations are recursive by nature, and use two further functions to ease the definitions. While the formal definition follows below, here is a short description of each of these functions:

- *inChar* determines the complete tuple of input data characteristics of a given activity.
- *outChar* determines the complete tuple of output data characteristics of a given activity.
- *singleIn* determines a single input data characteristic of a given activity, given its position in the operation signature. It is determined from the output data characteristic of the preceding activity.
- *combinedIn* determines the complete tuple of input data characteristics of a given activity, by combining the individual data characteristics determined through *singleIn*.

The latter two functions are related to, and make use of, the data flow edges of the workflow ($wd.\delta$), as described in Section 2.2.3.

$$\begin{aligned}
\text{inChar} : wd.A &\rightarrow \text{DC}\textsc{har}^n \\
a &\mapsto \begin{cases} D \text{ if } a = wd.s \\ \text{combinedIn}(a) \text{ otherwise} \end{cases} \\
\text{outChar} : wd.A &\rightarrow \text{DC}\textsc{har}^m \\
a &\mapsto \begin{cases} \text{inChar}(a) \text{ if } a \in \{wd.s, wd.e\} \\ \text{invokeChar}(\Omega(a), \Xi(a), inChar(a)).d \text{ otherwise} \end{cases} \\
\text{combinedIn} : wd.A &\rightarrow \text{DC}\textsc{har}^p \\
a &\mapsto (\text{singleIn}(a, 1), \ldots, \text{singleIn}(a, |a.I|)) \\
\text{singleIn} : wd.A \times \mathbb{N}_+ &\rightarrow \text{DC}\textsc{har} \\
(a_s, i) &\mapsto \text{outChar}(a_p)[o] \mid (a_p, o, a_s, i) \in wd.\delta
\end{aligned}$$

Inter-engine Data Transfer Constraints

Let $transferConstr$ denote a function which determines a co-allocation constraint of a single data transfer within the workflow, i.e., between two workflow engines orchestrating activities connected by a data flow edge. It returns a co-allocation constraint consisting of a single time-determining allocation constraint, encompassing the data transfer between the originating and receiving workflow engines. It uses the $transferTac$ function defined in Section 2.6.5:

$$\begin{aligned}\text{transferConstr} : wd.\delta &\rightarrow \textsc{Constraint}\\ (a_p, o, a_s, i) &\mapsto (\{\text{transferTac}(\text{engEndp}(\Xi(a_p)), \text{engEndp}(\Xi(a_s)),\\ &\quad transferOvhd + \text{size}(\text{outChar}(a_p)[o]))\}, \emptyset, \emptyset, \emptyset)\end{aligned}$$

In this definition, $transferOvhd$ denotes the overhead induced by the exchange protocol itself (e.g., SOAP Headers and Envelopes).

Workflow Input and Output Constraints

Let *inConstr* and *outConstr* denote functions which determine the co-allocation constraints related to the data transfers concerning the initial input to the workflow, and its final output. Both constraints consist of a single time-determining allocation constraint encompassing the bandwidth resources of the client machine and the workflow engine handling the workflow input/output, where the amount of data to be transferred is the sum of the sizes of all input/output data characteristics, plus the respective protocol overhead:

$$\begin{aligned}\text{inConstr} : \textsc{Wfs} &\rightarrow \textsc{Constraint}\\ ws &\mapsto (\{\text{transferTac}(u, \text{engEndp}(\Xi(wd.s)), inOvhd +\\ &\quad \sum_{i \in [1,|D|]} \text{size}(D[i]))\}, \emptyset, \emptyset, \emptyset)\\ \text{outConstr} : \textsc{Wfs} &\rightarrow \textsc{Constraint}\\ ws &\mapsto (\{\text{transferTac}(\text{engEndp}(\Xi(wd.e)), u, outOvhd +\\ &\quad \sum_{i \in [1..|\text{outChar}(wd.e)|]} \text{size}(\text{outChar}(wd.e)[i]))\}, \emptyset, \emptyset, \emptyset)\end{aligned}$$

Operation Invocation Constraints

Let $opConstr$ denote a function which, for all activities of the workflow, determines the corresponding co-allocation constraint graph:

$$\begin{aligned}\text{opConstr} : wd.A &\rightarrow \textsc{ConstraintGraph}\\ a &\mapsto \begin{cases}\emptyset \text{ if } a \in \{wd.s, wd.e\}\\ \text{invokeChar}(\Omega(a), \Xi(a), \text{inChar}(a)).cg \text{ otherwise}\end{cases}\end{aligned}$$

Co-allocation Constraint Graph of a Workflow Schedule

Both the assignments of operation instances (Ω) and of workflow engines (Ξ) indirectly lead to co-allocation constraints, or co-allocation constraint graphs. Furthermore, the

dependencies expressed through the workflow description, i.e., through $wd.\delta$ and $wd.\kappa$, lead to temporal dependencies between these constraints. All this information can be combined to define the co-allocation constraint graph of the workflow schedule itself as follows:

Definition 2.46. *Workflow Schedule Co-allocation Constraint Graph*
The co-allocation constraint graph wsg of a workflow schedule ws is the co-allocation constraint graph $wsg = (CC, <)$, as defined by the following properties:

1. CC contains the schedule's input and output constraints: $\text{inConstr}(ws) \in CC \wedge \text{outConstr}(ws) \in CC$
2. CC contains all co-allocation constraints imposed by the data flow edges: $\forall d \in wd.\delta : \text{transferConstr}(d) \in CC$
3. CC contains all co-allocation constraints imposed by the operation instance mapping: $\forall a \in wd.A \setminus \{wd.s, wd.e\} : \text{opConstr}(a).CC \subseteq CC$
4. CC contains no elements other than those defined in 1. – 3. above
5. The schedule's input constraint precedes all other constraints: $\forall cc \in CC \setminus \{\text{inConstr}(ws)\} : \text{inConstr}(ws) < cc$
6. The schedule's output constraint follows all other constraints: $\forall cc \in CC \setminus \{\text{outConstr}(ws)\} : cc < \text{outConstr}(ws)$
7. The control flow dependencies of the workflow description are preserved: $\forall (a_p, a_s) \in wd.\kappa : \text{opConstr}(a_p) \neq \varnothing \wedge \text{opConstr}(a_s) \neq \varnothing \Rightarrow \text{lastConstr}(\text{opConstr}(a_p)) < \text{firstConstr}(\text{opConstr}(a_s))$
8. The data flow dependencies of the workflow description are preserved with respect to the originating and receiving activities: $\forall (a_p, o, a_s, i) \in wd.\delta : (\text{opConstr}(a_p) \neq \varnothing \Rightarrow \text{lastConstr}(\text{opConstr}(a_p)) < \text{transferConstr}((a_p, o, a_s, i))) \wedge (\text{opConstr}(a_s) \neq \varnothing \Rightarrow \text{transferConstr}((a_p, o, a_s, i)) < \text{firstConstr}(\text{opConstr}(a_s)))$
9. Temporal dependencies expressed within co-allocation constraint graphs of individual operation invocations are preserved: $\forall a \in wd.A \setminus \{wd.s, wd.e\}, (CC_a, <_a) = \text{opConstr}(a), cc_p, cc_s \in CC_a : cc_p <_a cc_s \Rightarrow cc_p < cc_s$

□

The co-allocation constraint graph of a workflow schedule is completely determined by the schedule, and contains the aggregated information about the resource requirements of all contained activities (including sub-workflows) and data transfers, as well as the temporal relationship of these requirements. In contrast to the workflow definition alone, which is purely logical, the constraint graph is tied to the characteristic behavior – i.e., resource requirements – that a workflow execution will exhibit when run with a particular type of input, on the associated set of endpoints (workflow engines and operation instances).

In other words, wsg "fixes the schedule in space", and it contains all the information about temporal dependencies between the resource requirements. However, it does not "fix the schedule in time" yet. In fact, wsg again merely serves as a blueprint for the schedule's set of co-allocations ($ws.C$).

2.7.3 Workflow Schedule Validity and Reservation

For a workflow schedule ws to be valid, its co-allocations must be consistent with its co-allocation constraint graph. Furthermore, the validity of the schedule depends on the (reservation) state of the infrastructure, i.e., whether all its allocations are actually feasible in that state.

Definition 2.47. *Workflow Schedule Validity*
Let $ws = (wd, u, D, \Xi, \Omega, C)$ be a workflow schedule, and let wsg be the co-allocation constraint graph of ws. Further, let $S \in \text{STATE}$ be a reservation state, and let $t \in \mathbb{N}$ be a timestamp.

ws is said to be **valid** in state S at time t, if and only if all of the following hold:

- All of the co-allocations of ws are actually feasible in reservation state S at time t: $\text{feasible}(S, t, C) = \texttt{true}$
- There exists an injective function θ, mapping co-allocation constraints to co-allocations: $\theta : wsg.CC \rightarrow ws.C$
- θ associates every co-allocation constraint to a compliant co-allocation: $\forall cc \in wsg.CC : \theta(cc) \xrightarrow{\checkmark} cc$
- The temporal dependencies among co-allocation constraints are preserved in the associated co-allocations: $\forall cc_p, cc_s \in wsg.CC : cc_p < cc_s \Rightarrow \text{coAllocEnd}(\theta^{-1}(cc_p)) \leqslant \text{coAllocStart}(\theta^{-1}(cc_s))$

□

Just as multiple co-allocations may comply with a single co-allocation constraint, there is no single workflow schedule which is valid at a given time, for some workflow description and reservation state. In fact, there is either none, or an infinite number of valid schedules for any such combination. The former case occurs if no workflow engines are present, or if there is no operation instance for one of the workflow's activities. The latter claim is trivially explained by the fact that co-allocations could be infinitely "pushed" into the future.

The problem of determining a multiset of co-allocations which results in a valid workflow schedule is not a part of the formal model, but is discussed in detail in Chapter 3.

Reservations

If a workflow schedule ws is valid in reservation state S at timestamp t, then it is trivial to acquire the corresponding reservations R through the function *reserve* (which will also put the infrastructure in a new reservation state S^*):

$$S^* = reserve(S, t, ws.C)$$
$$R = S^* \setminus S$$

2.7.4 Workflow Schedules as Operation Instances

We have repeatedly stated that workflow definitions can be regarded as operations, and that every workflow engine we can be regarded as providing operation instances of any workflow description wd. While we will not go into detail about the actual orchestration here, because it is out of scope for this model, we will briefly describe how the metadata required for scheduling can be determined.

The operation $op \in \textsc{Op}$ associated with wd is defined as $op = \text{wfOp}(wd)$. For any workflow schedule $ws = (wd, u, D, \Xi, \Omega, C)$ and its associated co-allocation constraint graph wsg, one can define an operation instance $oi = (op, \text{engEndp}(we), ws)$.

The only metadata that must be determined from oi is its invocation characteristics, as determined via the *invokeChar* function, and can in be derived from the workflow schedule itself:

$$\text{invokeChar}(oi, u, D) := (\text{outChar}(wd.e), wsg)$$

This results in two simple observations: First, every workflow schedule corresponds to an operation instance, and directly represents "its own" invocation characteristics. Second, to determine a workflow schedule for a workflow description containing subworkflows, workflow schedules for all contained subworkflows must be determined as well. Intuitively, this corresponds to the "inlining" (or "flattening") of the contained subworkflows.

2.8 Summary and Discussion

The previous sections of this chapter presented a model which allows to formalize, describe, and reason on important aspects of an infrastructure in which the DWARFS system operates. To shortly recapitulate, the model presents a unified approach to represent:

- Resources, resource usage ((co-)allocations), resource requirements (co-allocation constraints), and their management across the infrastructure (reservation state)
- Operations and workflows, and instances thereof
- Methods to describe and predict the effects of operation invocations, concerning both the produced data and the required resources

- Workflow schedules

As the purpose of DWARFS is to provide *predictable* execution of workflows, workflow schedules present the most crucial and elaborate part of the model, combining all the other concepts. While the model, in and of itself, does not provide the concrete *means* (in terms of implementation) to do neither scheduling (Chapter 3) nor enforcement (Chapter 4), it does provide the necessary *foundation* for them in a concise manner.

In the following, we go into a bit more detail about a few aspects of the model which we deem noteworthy. This includes design choices, simplifying assumptions, and topics which may benefit from further elaboration.

General Assumptions about the Infrastructure and Resources

DWARFS aims to provide predictable workflow execution, based on resource management (which needs to be provided by the underlying infrastucture, which we will call DWARFS infrastructure for short). This in turn means that ideally, any and all resources within the system would be under full and exclusive control of DWARFS resource management. From a practical point of view, this is unfeasible in such absoluteness. For instance, DWARFS components will typically run as a simple (user-level) process on a computer, and will not be able to control system- or kernel-level scheduling. Likewise, there may be other processes running on such machines which require network resources beyond DWARFS' control. However, we assume that such resource usage which is not under the control of DWARFS is negligible in the sense that it does not adversely affect DWARFS resource management.

Concerning bandwidth resources, we treat incoming and outgoing bandwidth of a bandwidth as completely separate resources (assuming full duplex connections), while in reality there is normally some interference – for example, an outgoing TCP connection still generates some control data on the incoming channel. This is one example which falls under the abovementioned simplification. Similarly, for network transfers (which require co-allocations of multiple resources), we ignore the minimal network latencies introduced by the physical connections.

The model also ignories latencies caused by the access to persistent resources such as storage. While it is true that at the time of writing, most consumer-grade network devices provide faster throughput than could be written to consumer-grade storage (thus rendering co-allocations of incoming bandwidth and storage, which require data to be piped to storage, impossible), this is not an intrinsic property of transitory or persistent resources, or of the model. As a counter-example, enterprise-level storage hardware is currently able to deliver I/O rates of 6000/4400 MB/s, which outperforms 10GB-Ethernet [San13]. Therefore, we argue that such considerations would only render the model needlessly complex.

Provider-side Definition of Contraints

One noteworthy aspect of the DWARFS system model is that essentially all resource requirements (co-allocation constraints) are dictated by the infrastructure, i.e., by the

resource providers, instead of by the users submitting workflows. This is in sharp contrast to traditional Grid environments. While we will go into more detail in Chapter 7, such environments generally require the users to state their requirements (e.g., "This task will require a 64-bit quad-core machine with 16 GB of RAM for 3 hours"). This is caused by the nature of the environment itself: in Grid infrastructures, users are in fact submitting their own programs to be executed on remote machines, and will therefore need to specify their requirements themselves.

However, we argue that in a truely loosely coupled SOA, the actual implementation is procured directly by the provider. Thus, details of an operation also need only be known by the provider: users must know *what* an operation does, but not *how* it does it.

Workflow Descriptions

DWARFS is specifically addressing the needs of Scientific Workflows with possibly very large data flows. Like other SWF Systems (e.g., [OAF$^+$04, STS$^+$06]) it uses Directed Acyclic Graphs (DAGs) as the basis of a workflow description's structure. DAGs are suitable for many (albeit not all) workflows which are defined based on the contained data flow, however they lack sophisticated control flow possibilities such as loops. One key feature of DWARFS is the use of Advance Reservations, where a workflow schedule predicts (future) resource requirements both in terms of location (*which* resource), as well as concrete time intervals (*when*) and usage (*how much*). Complex control flow decisions inside workflow descriptions, such as arbitrary cycles or conditional branches hamper the predictability we strive for, because the exact execution flow of the workflow is normally not predictable from the workflow description alone.

A minor simplification concerns data types and data type compatibility: workflow descriptions only allow data transfers (i.e., assignments of the output of one activity to an input of a successor activity) if the data types strictly match. While the model can be extended with a full-fledged type system including subtyping and compatibility checks, we consider such extensions as out of scope for this thesis.

Data Characteristics and Co-Allocation Constraints

We are fully aware that a characterization (or classification) of concrete data, i.e., partitioning of the domain of a data type into meaningful equivalence classes, for any non-trivial domain, may be difficult or even impossible. Still, DWARFS requires such (domain-specific) characteristics to be available and as fine-grained as possible, including the possibility of estimating upper bounds of the size of such data. Similarly, the runtime behavior (i.e., resource requirements) of some operation instances may be difficult to predict. Such problems might be attenuated by employing statistics and/or data mining and machine learning techniques (cf. Section 7.3). While there may be application domains or implementations for which the required metadata cannot be determined with sufficient precision, we are confident that there are many others where either analytic or empirical approaches can produce such metadata.

The quality of the available metadata directly influences the quality of the scheduling, and thus of the entire DWARFS approach. As shown in Figure 2.13, one can in fact regard the "usefulness" of our approach as a function of the quality of the provided

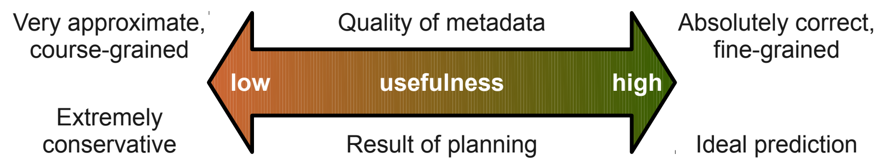


Figure 2.13: DWARFS predictability: Metadata Quality and Scheduling Results

metadata. On one end of the spectrum, metadata is extremely precise, therefore producing high-quality schedules. On the other end, if the provided metadata contains very little or low-quality information, the scheduling can only return extremely conservative results. While our approach is indeed of little use in the latter scenario, we do assume that metadata can be produced with sufficient quality, so that there is indeed a gain in predictability. Even so, and as we shall cover in more detail in Chapters 4 and 6, we do not necessarily expect, or require, predictions to be totally accurate. Rather, considering a safety margin, or "erring on the safe side" by slightly overestimating the predicted resource requirements allows to compensate for the fluctuations inherent to complex and dynamic systems, while still preserving the benefits of predictability.

3

Planning Under Quality of Service Criteria

The previous chapter gave an exhaustive overview about the main ideas and concepts behind DWARFS, and introduced a formal model which captures all the relevant notions. It concluded with the statement that predictability of workflow execution is one of the most important assets of DWARFS. Such predictability is achieved through proactive resource management by means of Advance Reservations, which requires workflows to be scheduled prior to their execution.

This chapter details our approach to scheduling workflow executions in DWARFS. We start by examining the degrees of freedom that exist when determining valid workflow schedules for a workflow description, and various possible (combinations of) optimization goals. We then describe our optimization approach, which is based on a Genetic Algorithm (GA), focusing on noteworthy details concerning the implementation of the chromosome, and fitness functions. Finally, we show how our approach deals with two particular aspects of the planning, namely the partitioning of the workflow orchestration among several co-operating workflow engines, and the optimized handling of data transfers.

3.1 QoS Metrics and Goals

Any workflow which is executed in a DWARFS infrastructure makes use of resources which must have been reserved before they can be utilized. It is therefore mandatory to plan the execution in advance – in other words, to determine a valid workflow schedule which fits the user requirements, and to make the corresponding reservations. In simple terms, the goal of this planning (or scheduling) phase is thus to evaluate a number of possible schedules with respect to their suitability regarding the user's criteria. Before going into detail about what these criteria may be, how to express them, and how to evaluate workflow schedules according to the criteria, let us take a general look at what possible candidate schedules look like, and how they differ.

There are three components of a workflow schedule – the workflow definition wd, the invoking endpoint u, and the input data characteristics D – which are supplied by the user, and thus fixed and shared by all candidate schedules. The remaining components (Ξ, Ω, and C) can be varied in the following ways:

- The workflow engine mapping Ξ determines which activity of wd is orchestrated by which workflow engine. In principle, any available workflow engine can be in charge of handling any activity (with the exception of the start and end activity having to be located on the same workflow engine). There are thus $|\textsc{Engine}| \times (|wd.A| - 1)$ possibilities of assigning workflow engines to activities.
- The operation instance mapping Ω determines which operation instance executes each activity. The total number of possible assignments is the product of the number of operation instances for each workflow activity.
- The co-allocations C are influenced both by the co-allocation constraint graph (which in turn depends on the choice of Ξ and Ω), and by the infrastructure's reservation state. In principle, there is an infinite amount of possibilities to choose co-allocations (while still producing valid schedules), because co-allocations can potentially be shifted arbitrarily far into the future.

If there is an infinite number of candidate solutions for every input, then which solution is "best" for the input at hand? Put simply, one needs to be able to evaluate the candidates with respect to the user's requirements. This in turn leads to two further questions: how can such requirements be expressed, and what are suitable metrics to evaluate a schedule's conformance?

Every workflow schedule ws refers to a set of co-allocations which are required for its execution. There are three very useful metrics that can immediately be derived from a workflow schedule with the system model, as outlined below.

Workflow Schedule Termination

The termination timestamp of a workflow schedule is obtained through the function *wsTermination*:

$$\begin{aligned} \text{wsTermination} : \textsc{Wfs} &\rightarrow \mathbb{N} \\ ws &\mapsto \max_{c \,\in\, ws.C}(\text{coAllocEnd}(c)) \end{aligned}$$

Workflow Schedule Duration

The duration of a workflow schedule is obtained through the function *wsDuration*:

$$\begin{aligned} \text{wsDuration} : \textsc{Wfs} &\rightarrow \mathbb{N} \\ ws &\mapsto \text{wsTermination}(ws) - \min_{c \,\in\, ws.C}(\text{coAllocStart}(c)) \end{aligned}$$

Workflow Schedule Cost

The cost of a workflow schedule at a given timestamp is obtained through the function *wsCost*:

$$\begin{aligned} \text{wsCost} : \textsc{Wfs} \times \mathbb{N} &\rightarrow \mathbb{N} \\ (ws, t) &\mapsto \text{orchCost}(ws, t) + \sum_{c \,\in\, ws.C} (\text{coAllocCost}(c, t)) \end{aligned}$$

Herein, *orchCost* refers to the additional cost imposed by the system orchestrating the schedule. In other words: the *coAllocCost* function determines the cost of resources which are required to execute the individual steps of the workflow; these costs are defined by the individual operation providers (and for data transfers, the network infrastructure). In fact, the respective providers need not even be aware of these resource usages happening within a workflow. Conversely, the orchestration cost represents the costs associated with the coordinated execution of the workflow itself, and covers the additional costs imposed by the involved workflow engines, as well as other factors (for instance, one could imagine an additional "commission" fee for the orchestration, etc.)

Other Metrics

The abovementioned metrics can immediately be derived from the system model. It is of course possible to define arbitrarily complex other metrics, for example to take into account the reputation of operation instances (e.g., using availability statistics), their environmental footprint, etc., by extending the model accordingly. For this thesis, we chose to stay with the above metrics.

Degrees of Freedom and their Effects on Metrics

The following intuitive samples demonstrate how the degrees of freedom present for the assignments of Ξ, Ω and C may influence the abovementioned metrics:

- **Chosen providers for operations:** There are possibly many providers offering the operations required for every single activity within the workflow, differing in the resulting cost, and execution time.
- **Operation instances' resource availability:** Every operation invocation needs to be able to reserve the required resources, as specified by the provider. However, because the infrastructure keeps track of already allocated resources, not all theoretically possible reservation requests are actually feasible (or are only feasible at a later time, thus delaying the execution).
- **Resource allocation requests:** While providers state which resources are required and give bounds on their usage, this information may allow for flexibility. As shown previously, co-allocation constraints may be modified (i.e., restricted) to weigh execution time against cost.

- **Workflow Engine instance for activity:** Since we are dealing with a distributed execution engine, every workflow activity may be handled by any available workflow engine in the system. This choice again affects the overall planning, as for example network throughput (and availability) between different engines and operation providers differ.

- **Timing:** Because of the abovementioned aspects – especially those on the availability of resources – even small changes in the planned timing may substantially influence the overall outcome. As an example, starting the workflow execution a few seconds later may be able to use resources at an operation instance that was previously fully loaded, and thus actually result in a faster overall execution time.

Optimization Goals

The abovementioned metrics allow to evaluate and compare multiple schedules with respect to the corresponding aspects – like runtime, or cost – of the schedules. From there, it is only a small step to actually formulate user requirements for the planning – for example, by specifying the optimization goal to be the minimization of a particular metric. DWARFS supports a slightly more intricate form of those requirements, which may include multiple aspects of multiple metrics, as described below.

- **Single-objective optimization**, e.g., "Execute the workflow as fast as possible". This is arguably the most common and traditional kind of optimization, where the goal is to minimize (or maximize) a particular metric. The suitability of every schedule is proportional to the metric to optimize.

- **Single-objective bound**, e.g., "Finish before 5 pm". This is somewhat similar to the above in that the goal is to minimize a single metric, but only to a specific threshold, beyond which all conforming schedules are considered equally acceptable. On the other hand, and unlike the previous goal, any schedule which does not reach the threshold is considered completely unsuitable. In other words, such bounds constitute hard constraints and correspond to a binary decision, marking candidates as either suitable or unsuitable.

- **Multi-objective optimization**, e.g., "Execute as fast as possible *and* terminate as early as possible". The goal here is to minimize/maximize multiple metrics at the same time.

- **Multi-objective optimization with multiple bounds**, e.g., "Balance execution speed and cost, but still finish before a given deadline and stay within a given budget".

In fact, the last requirement in the above list is the most general kind of statement, of which the others are merely special cases. A detailed description of our method to achieve and implement such optimization goals will be given in Section 3.4.5.

3.2 Optimization approach

Finding the best schedule according to such user requirements ultimately resolves to an optimization problem. There are two important observations to be made here: First, as seen from the examples of the specification of QoS criteria, we are in fact dealing with a *multi-objective optimization problem*, whereby multiple, and possibly conflicting, goals, may be specified and must be accounted for. Second, there is an extremely large search space, which renders exhaustive search or analytical methods infeasible – especially given the generally unpredictable nature of the existing reservation state at planning time.

We have therefore opted for a metaheuristic approach, namely a Genetic Algorithm (GA). Genetic algorithms have been applied to many optimization domains – in particular to scheduling problems – and are well-described in the literature (e.g., [Gol89, Mit98], as well as the Related Work in Chapter 7). Therefore, we will limit ourselves to a short introduction of the overall concept here; we will however go into more detail on select aspects of our implementation, where such explanations are useful or required.

3.2.1 Genetic Algorithms in a Nutshell

Genetic algorithms use an optimization approach that mimics natural selection as it happens in the real world: Each possible solution to the problem to be solved is represented as a genotype (or individual), represented by a single chromosome. The chromosome consists of a number of genes which can have different values (allele). Finally, a population consists of a number of individuals with different gene expressions. A fitness function is used to assign each individual within the population a value which determines its suitability in reaching the optimization goal.

The actual optimization then takes place by evolving the population into a new generation of individuals. For finding new problem solutions, two approaches are generally employed: mutation (i.e., randomly changing the value of a random gene), and crossover (mating of two individuals by recombining their genes). As these operations increase the number of genotypes, the fitness function is used to select the best individuals – considering both the originally existing individuals, and the newly created ones – and carry them over to the next generation (survival of the fittest).

Consider the following (extremely simple) optimization problem: "Find the largest unsigned 8-bit integer". Naturally, this is a problem which does not actually require any meta-heuristic approach at all, because the solution is known in advance. However, it is still useful to introduce the concepts and approaches of a GA.

Chromosome representation

Each potential solution must be encoded as a chromosome, consisting of a number of genes. A widely used approach is to use arrays of simple types – in this case, an array of 8 bits. Thus, the array itself (`bit[8]`) represents the chromosome, while each array

element (`bit`) represents a gene, and each element's value (`0` or `1`) represents an allele. In this example, all genes share the same type, but this is not generally the case.

Population and Fitness Functions

A population is merely a number of individuals (each consisting of a single chromosome). At the start of the optimization, all individuals are assigned random alleles. A fitness function assigns each individual a single numeric value, representing its suitability for the problem at hand. The fitness function is crucial in that it is meant to "represent" the optimization goal. By convention, larger fitness values indicate better suitability. In this simple case, the most obvious fitness function could be defined as:

$$
\begin{aligned}
fitness : \{0,1\}^8 &\rightarrow \mathbb{N}_+ \\
(b_7, \ldots, b_0) &\mapsto \sum_{0 \leqslant i \leqslant 7} b_i \cdot 2^i
\end{aligned}
$$

For an entire population, one then simply applies the fitness function to every individual. The individual with the highest fitness score is the one that fulfills the optimization goals best. Figure 3.1 shows a sample population with the associated individual fitnesses.

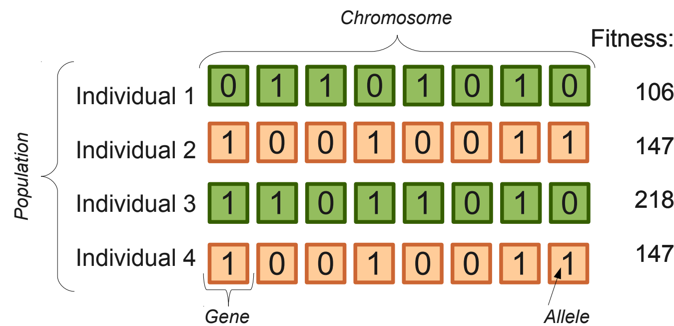


Figure 3.1: Genetic Algorithm: Sample Population and Fitnesses

Mutation and Crossover

In order to find new solutions to the problem, new populations (or generations) are created by producing new individuals. This is achieved by applying genetic operators to existing individuals. The most common operations employed are mutation or and crossover (also called recombination), both of which are depicted in Figure 3.2. Note that neither of these operations is guaranteed to produce fitter individuals. The purpose of them is to produce genetic diversity, which, in combination with the natural selection explained below, will *eventually* lead to the appearance of fitter individuals, corresponding to more suitable solutions to the optimization problem.

Mutation In mutation, a new individual is produced by randomly varying a (random) number of alleles of an existing individual.

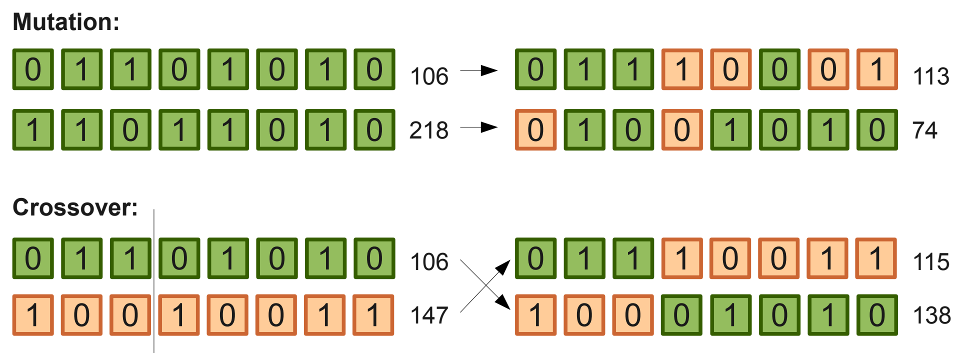


Figure 3.2: Genetic Algorithm: Mutation and Crossover

Crossover In crossover, two existing individuals are "mated" by recombining their genes. When the chromosome is represented as an array of elements, the simplest possible form of crossover is achieved by splitting each array at the same random position, and recombining the fragments into two new arrays.

Natural Selection

Both mutation and crossover can produce new individuals from existing ones. For choosing *which* individuals of a generation are actually considered for reproduction (i.e., the creation of a new generation), multiple strategies are possible. We quickly describe some commonly used ones below:

- Random selection: In this kind of selection, all individuals of a population have the same probability of being selected for breeding a new generation. In other words, there is no preference, neither for fit nor for unfit individuals.
- Truncation selection: The population is sorted according to the individuals' fitnesses, and only a predefined number (or percentage) of top-ranked individuals is chosen for reproduction. Practically, individuals with low fitnesses thus have no chances to breed.
- Roulette wheel selection: The probability of an individual of being selected for breeding is directly proportional to its fitness (relative to the fitnesses of the entire population). This strategy has a tendency to prefer fitter individuals for breeding (thus generally speeding up convergence), but without completely neglecting less suitable individuals (thus allowing to "escape" local optima).

Evolution

For producing a new generation from an existing one, selection and genetic operators are repeatedly performed until enough new individuals have been produced. In a slight variation of this strategy called *elitism*, a fixed number of the top-ranking solutions of the original population are carried over to the next generation unchanged (in addition to the breeding of new individuals).

3.3 Chromosome Representation

During the planning, each candidate schedule is represented as a GA individual, i.e., as a chromosome consisting of multiple genes. Figure 3.3 shows an example chromosome for a trivial workflow consisting of a single operation invocation.

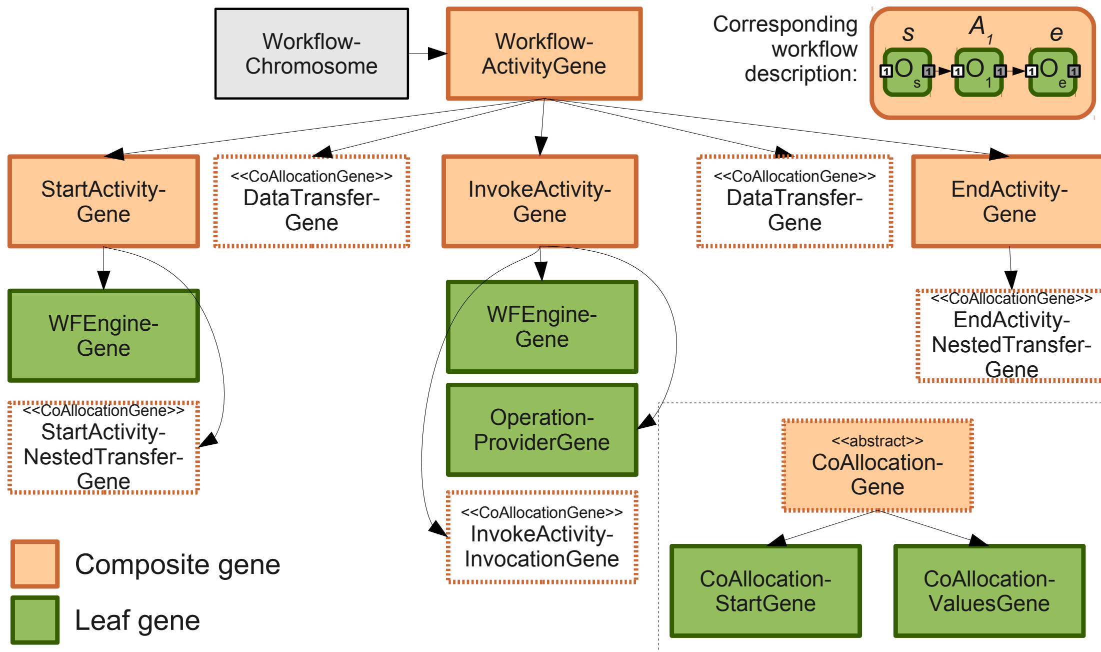


Figure 3.3: Sample Workflow Chromosome Layout

An immediate observation is that this chromosome does not merely contain a sequence of genes, but a tree of nested genes instead. Such a representation offers multiple advantages. First, at the highest level, it allows for chromosomes to roughly resemble the structure of the workflow definition that they represent, thus easing the comprehension of the semantics of the chromosome. Furthermore, the composition of genes reflects their logical structure in terms of several degrees of freedom which are independent, but together influence one particular aspect (e.g., an invocation depends on the operation provider, the workflow engine, and the co-allocations). Finally, such a modular design allows to re-use individual (sub-) genes in different contexts.

3.3.1 Genes

In this section, we introduce the various genes that we use to represent a workflow schedule as a chromosome for a Genetic Algorithm. Generally speaking, there are two types of genes: composite genes (which are only composed of sub-genes) and leaf genes. In simple terms, leaf genes contain the actual mutable values (i.e., the alleles), while composite genes merely present a (functional) aggregation of their subgenes.

WorkflowActivityGene

The `WorkflowActivityGene` is a single top-level gene which represents an entire workflow. Its structure – i.e., the contained sub-genes – directly depends on the workflow description.

StartActivityGene

This gene represents the explicit entry point of the process, i.e., the *s* operation of the workflow description. It is a composite gene which contains a `WFEngineGene`, representing the workflow engine with which the user is interacting to start the workflow execution, and a `StartActivityNestedTransferGene`, representing the initial data transfer from the client host to that workflow engine.

EndActivityGene

This gene represents the explicit exit point of the process, i.e., the *e* operation of the workflow description. Unlike the `StartActivityGene`, it does not contain a `WFEngineGene`; rather, the workflow engine of the `StartActivityGene` is reused. This ensures that the workflow start and end are actually scheduled at the same workflow engine, so that the SOA semantics of "a workflow invocation is perceived as just another Web Service" can be maintained. This gene contains an `EndActivityNestedTransferGene`, representing the final data transfer of the workflow output to the client.

StartActivityNestedTransferGene and EndActivityNestedTransferGene

These genes represent the transfer of the input and output data from/to the end user (in terms of a co-allocation of bandwidth resources), and are subclassing the CoAllocationGene (described below). The co-allocation constraints – which resources are needed, how large is the data transfer, which throughput range is acceptable – are obtained from the respective parent gene, and mutated by the subgenes of the CoAllocationGene itself. The actual co-allocations are determined from those mutated co-allocation constraints.

WFEngineGene

This gene represents the workflow engine executing the activity denoted by its parent gene. The possible allele values are in principle all workflow engines known to exist in the infrastructure. In the current implementation, workflow engines are simply identified by their SOAP endpoint URL.

DataTransferGene

These genes are inserted whenever data produced by one activity is required as input by another activity. In other words, they represent the data flow edges in the workflow description. Just like, for instance, the `EndActivityNestedTransferGene`, they rep-

resent data transfers, in this case between two activities (i.e., between the workflow engines of two activity genes which are siblings of the DataTransferGene).

ControlTransferGene

These genes represent the control flow edges in the workflow description. They can be logically considered as the simplest form of data flow, where no data is actually transmitted (and thus no allocations are required). In fact, a `ControlTransferGene` merely serves as a way to express the precedence relationships between two activity genes in the same way as a `DataTransferGene`, but does not contain any mutable parts.

InvokeActivityGene

This is a composite gene that models a remote operation invocation. It consists of the subgenes `WFEngineGene`, `OperationProviderGene`, and `InvokeActivityInvocationGene`.

OperationProviderGene

This gene "chooses" an operation instance for the operation that its parent gene represents. Possible allele values are all available operation instances of the particular operation. Similarly to workflow engines, operation instances are identified by their SOAP endpoint URL.

InvokeActivityInvocationGene

This gene represents the actual invocation of an operation, in terms of co-allocations for the entire operation invocation (upload, execution, and download). It has dependencies on its parent and sibling genes for determining the allocation constraints.

CoAllocationGene

All genes that are shown in white dotted boxes are actually subclasses of this gene. The subclasses are responsible for providing co-allocation constraints, while the CoAllocationGene itself is responsible mutating these constraints (i.e., to produce co-allocation constraint restrictions), and for actually finding compliant co-allocations. Mutation is performed by leveraging the subgenes described below; the method for finding co-allocations is detailed in Section 3.3.6.

CoAllocationStartGene

This gene simply consists of a number influencing the delay when finding co-allocations; in other words, this value allows to shift the timestamp, at which the *search* for compliant co-allocations starts, into the future.

CoAllocationValuesGene

This gene contains an allele that represents the variable part of the coallocation to be found. It is actually an array of values which can be mutated independently (so strictly speaking, contains more than one mutable part). The values herein affect the minimum and maximum values to request for transitory resources that must be allocated.

3.3.2 Correspondence with the Formal Model

A `WorkflowChromosome` is the entity which implements a workflow schedule as defined in Section 2.7. In fact, its structure directly represents the workflow description, while its contents (the alleles) represents the schedule itself.

Workflow Description

A workflow description defines both the structure of the workflow – i.e., the activities it consists of – and their dependencies in terms of data and control flow. Both of these are directly reflected in the structure of the corresponding chromosome. In particular, this means that for any workflow description wd, the corresponding chromosome is constructed so that:

- every activity in $wd.\omega$ is represented by an `InvokeActivityGene` (for "normal" operations), a `WorkflowActivityGene` (for sub-workflows), or a `StartActivityGene`/`EndActivityGene` (for the start and end activities s and e)
- every data flow edge in $wd.\delta$ is represented by a `DataTransferGene`
- every control flow edge in $wd.\kappa$, which is not caused by a data flow edge, is represented by a `ControlTransferGene`

As the data and control flow edges constitute dependencies within the workflow execution, the corresponding genes (siblings in the tree representation of the chromosome) are also "wired" together to represent these dependencies. While details are further elaborated in Section 3.3.3, let us take a high-level look at where such dependencies are encountered, and the implications on the chromosome structure.

Figure 3.4 shows the same workflow description and chromosome as Figure 3.3, but from a different perspective: details about the composite genes have been mostly omitted; instead, this figure shows the dependencies representing control and data flow that need to be observed between genes – in other words, these genes "know" that they have a relationship with other genes. These dependencies can be directly extracted from the DAG representation of the workflow description.

Figure 3.5 shows a more complicated workflow description fragment (in the interest of readability, non-essential details have been omitted). This fragment shows how a chromosome is organized in the presence of parallel workflow steps[1]. Every gene representing an activity has dependencies on exactly as many transfer genes as there are

[1]We use the term *workflow step* as a uniform way to reference both activities and data or control transfers.

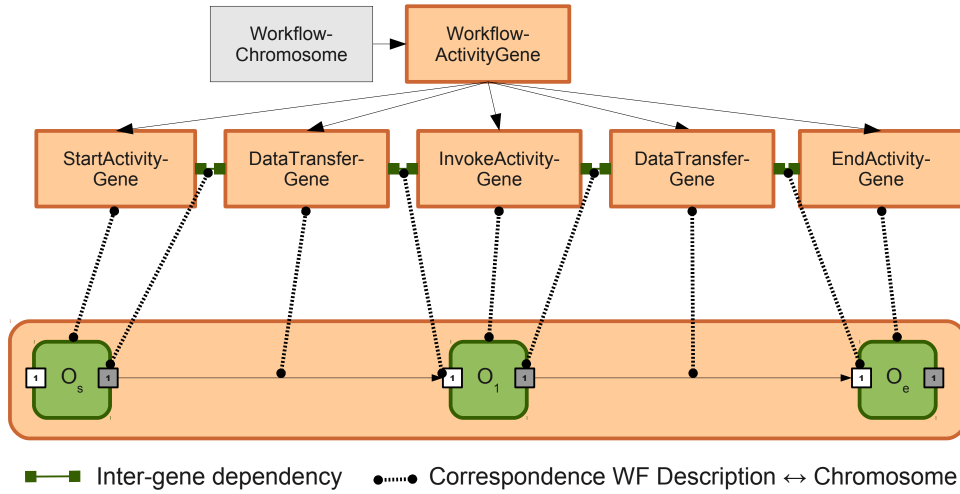


Figure 3.4: Correspondence of Workflow Description and Chromosome layout (Simple Workflow)

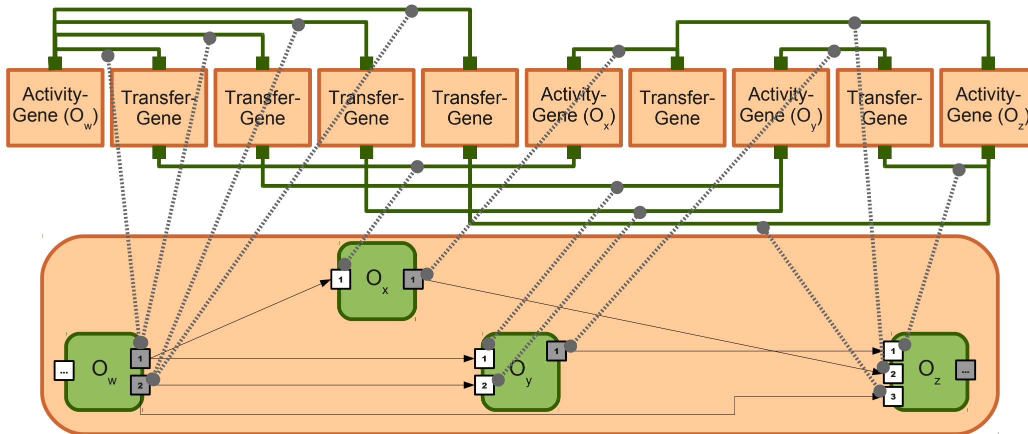


Figure 3.5: Correspondence of Workflow Description and Chromosome layout (Complex Workflow Fragment)

control and data flow edges targeting that activity. Similarly the transfer genes depend on the genes representing the source activity of their associated edge in the workflow description.

Moreover, as can be seen in the figures, the chromosome structure itself also reflects the workflow description structure – the order of the genes corresponds to a topological sort of the workflow graph.

Workflow Schedule

The variable components of a workflow schedule are the workflow engine mapping Ξ, the operation instance mapping Ω, and the co-allocations C. All of these are present in the alleles of the chromosome, as expressed in the `WFEngineGene`, `OperationProviderGene`, and `CoAllocationGene` genes. While the values of Ω and Ξ are straightforward to derive from the individual genes (they correspond directly to the contained alleles), the determination of the co-allocations is more involved, and is further discussed subsequently.

3.3.3 Interpretation of the Chromosome

Every gene within the chromosome can be mutated independently, and every possible allele combination results in a valid schedule (even if it may still be considered unsuitable with regard to the envisaged QoS objectives, e.g., because it takes longer than a user-specified deadline bound).

However, the alleles only denote the most basic information that is required to deduce the characteristics of the schedule that the individual represents. In fact, the actual interpretation of the genotype (answering questions such as: "when does it terminate, how much does it cost, which co-allocations are required?") is rather complex.

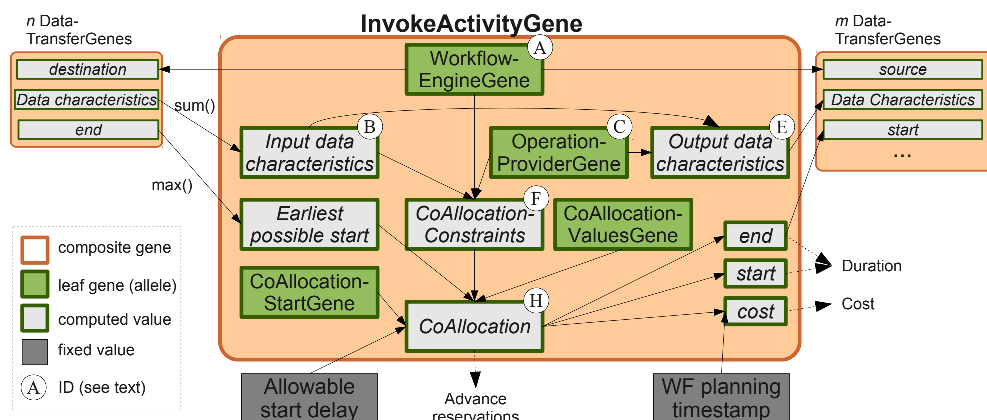


Figure 3.6: Chromosome Interpretation and Interdependencies

For a representative example, consider Figure 3.6, which presents a close-up look into the information gained from an `InvokeActivityGene`, and its relations with its predecessor and successor genes.

The `InvokeActivityGene` itself is depicted as the central box, where its sub-genes (or rather: the information obtained from the alleles of its sub-genes) are shown using filled green boxes. These entities are exactly those depicted in the central column of Figure 3.3, albeit using a different representation. The figure further shows a set of computed values (gray boxes with green outline) and some static values (dark gray boxes)[2]. The arrows depict dependencies between these values.

Before we go into more detail about the dependencies and calculations "within" this particular gene, consider the dependencies between the genes (i.e., relationships with preceding and successive workflow steps). It is relatively straightforward to see that those dependencies are again reflecting the structure, as well as integrity constraints, of the workflow. For example, the earliest possible start for the activity is the maximum of the timestamps when all of its preceding workflow steps have finished. All other genes have a similar structure. Thus, when looking at the chromosome as a whole, we again see a directed acyclic graph of dependencies, directly stemming from the workflow description, but at a much more fine grained level: at the workflow description level, dependencies are visible between workflow steps, while at the chromosome level, we see dependencies between the concrete values and calculations of which the workflow steps consist. This graph is spanning all of the alleles within the chromosome: for instance, in order to determine the end timestamp of the complete schedule – corresponding to the end timestamp of its `EndActivityGene`, all of the alleles of the entire chromosome have to be inspected, and essentially all computations within the graph need to be performed.

Thus, ultimately, all user-relevant information of a (complete) workflow schedule can be determined by examining its co-allocations. This is trivially true for the co-allocations themselves (which will eventually be reserved), but it also holds for the metrics, which are functionally dependent on the co-allocations. A condensed overview about how the alleles and intermediate calculations shown in Figure 3.6 are interpreted to determine the associated co-allocation, and how they relate to the model presented in Section 2, is given in Table 3.1. More detailed descriptions are given below.

ID	Domain	Source / calculation	Properties / misc.
A	ENGINE	`WorkflowEngineGene`	
B	DCHAR^n	*Input data characteristics*	
C	OPINST	`OperationProviderGene`	
D	INVCHAR	invokeChar($C, A.ep, B$)	$D.cg$ consists of a single constraint
E	DCHAR^m	$D.d$	Assigned to *Output data char.*
F	CONSTRAINT	firstConstr($D.cg$)	
G	CONSTRAINT	F "transformed" by `CoAllocationValuesGene`	$G \trianglelefteq F$
H	COALLOC	(see text)	$H \xrightarrow{\checkmark} G \Rightarrow H \xrightarrow{\checkmark} F$

Table 3.1: Chromosome Interpretation and Relation to Model

[2]These values are considered static in the sense that they do not change during the entire optimization run.

- $\boldsymbol{A} \in$ **ENGINE** – The workflow engine that is used to perform the operation invocation is directly determined from the allele of the `WorkflowEngineGene`.
- $\boldsymbol{B} \in$ **DCHAR**^{n} – The input data characteristics for the invocation are aggregated from the n `DataTransferGenes` that this `InvokeActivityGene` depends on.
- $\boldsymbol{C} \in$ **OPINST** – The operation instance used to perform the operation invocation is directly determined from the allele of the `OperationProviderGene`.
- $\boldsymbol{D} \in$ **INVCHAR** – The invocation characteristics for this particular combination of workflow engine, input data characteristics, and operation instance are calculated as $D = \text{invokeChar}(\text{C}, \text{A.ep}, \text{B})$. Because the operation is a simple operation (i.e., not a sub-workflow), its co-allocation constraint graph consists of a single co-allocation constraint. Thus, $\text{firstConstr}(D.cg) = \text{lastConstr}(D.cg)$.
- $\boldsymbol{E} \in$ **DCHAR**^{m} – The output data characteristics of this invocation are derived from the invocation characteristics: $E = D.d$.
- $\boldsymbol{F} \in$ **CONSTRAINT** – The co-allocation constraint that must be satisfied for this invocation is the single co-allocation constraint contained in the invocation characteristics: $F = \text{firstConstr}(D.cg)$.
- $\boldsymbol{G} \in$ **CONSTRAINT** – The `CoAllocationValuesGene`'s allele contains a set of values that influence the mutations performed on the co-allocation constraints for this invocation. Put simply, the *min* and *max* values of all time-determining allocation constraints contained in F are modified to produce restrictions of these constraints. The resulting co-allocation constraint is thus a restriction of the original co-allocation constraint: $G \trianglelefteq F$.
- $\boldsymbol{H} \in$ **COALLOC** – For the overall schedule to be correct, a co-allocation H needs to be found which i) complies with the co-allocation constraints ($H \xrightarrow{\checkmark} G$), and ii) observes the temporal dependencies of the workflow. To satisfy the latter condition, the co-allocation must not start before all of the activity's predecessor workflow steps have ended, denoted as *Earliest possible start* in the figure, and referred to as e from here on. This value is further modified by the allele of the `CoAllocationStartGene` ($m \in \mathbb{R} \mid 0 \leqslant m \leqslant 1$) and the *Allowable start delay* (d). After finding the co-allocation H (as described in Section 3.3.6), the following properties hold: $H \xrightarrow{\checkmark} G \wedge \text{coAllocStart}(H) \geqslant e + m \times d$

3.3.4 Chromosome as a Variable Dependency Graph

As stated in the previous section, to correctly interprete the information contained within a chromosome, one needs to evaluate the alleles contained within the genes, and perform intermediate calculations, which depend on allele values and/or other intermediate calculations. Internally, we use the unified notion of **Variable** to refer to these values, whether depending on an allele, or some other calculations. A variable is generic (in the sense of being able to "wrap" values of any data type), and has the following properties:

1. It is able to determine which other variables it depends on (called its dependees)
2. It is able to register itself to all its dependees
3. It has an internal ID which is greater than all its dependees' IDs
4. It keeps references to all registered dependent variables (called dependers)
5. It keeps its own calculated value cached
6. It can (re-)calculate its own value explicitly on demand, or implicitly (when requested by a depender but no value is cached yet)

Alleles are then simply associated with special variables which do not depend on other variables, but on the allele itself, and whose value merely reflects the allele.

By using only the properties 1. and 2., it is straightforward to construct a **Variable Dependency Graph** (VDG) at runtime (making sure that properties 3. and 4. are observed at all times). As a result, the identifiers of the contained variables are topologically sorted. Figure 3.7 shows a VDG, as obtained from the chromosome depicted in Figure 3.4. Note that unlike in most other figures, the arrows depicting relationships point from dependers to dependees (thus better visualizing the properties 1. and 2. above).

Figure 3.8 provides still further detail, by zooming in on a part of such a VDG, showing an example of what the individual values contained in the variables might look like, and how exactly they relate to each other.[3] In order not to overload the figure, a few details are still omitted, and the formalism is slightly simplified. The variables with IDs 1, 2, and 3 (termed `v1` – `v3`) contain the data characteristics of the user input to the workflow, which is also the sole input to the operation O_1, and the workflow engine and operation instance executing O_1. `v4` represents the invocation characteristics of the corresponding invocation, and `v14` is the co-allocation constraint contained within `v4`. This co-allocation constraint is shown in the lower right of the figure (consisting of time-determining allocation constraints for data upload, execution, and data download). `v30` contains the timestamp after which the co-allocation satisfying the constraint (`v31`) must be found. Finally, there are two variables which mutate the co-allocation constraint itself (`v15`) and the earliest-start timestamp (`v16`). The co-allocation constraint shown in the bottom right shows both the original resource constraints as obtained from `v14` (striked through), and the final restricted values, after applying the mutations contained in `v15` (in bold).

Such a VDG contains all the required information to determine a complete workflow schedule (i.e., all co-allocations and all metrics). It is the most low-level representation that we use. The following list summarizes the various levels of abstraction that we have described so far, shows how they relate to each other, and briefly discusses their properties.

1. **Workflow Description**: The workflow description contains the high-level definition of the workflow. Dependencies are expressed through the data and control flow edges.

[3]The figure presents a "hand-made" example – the values are only illustrational.

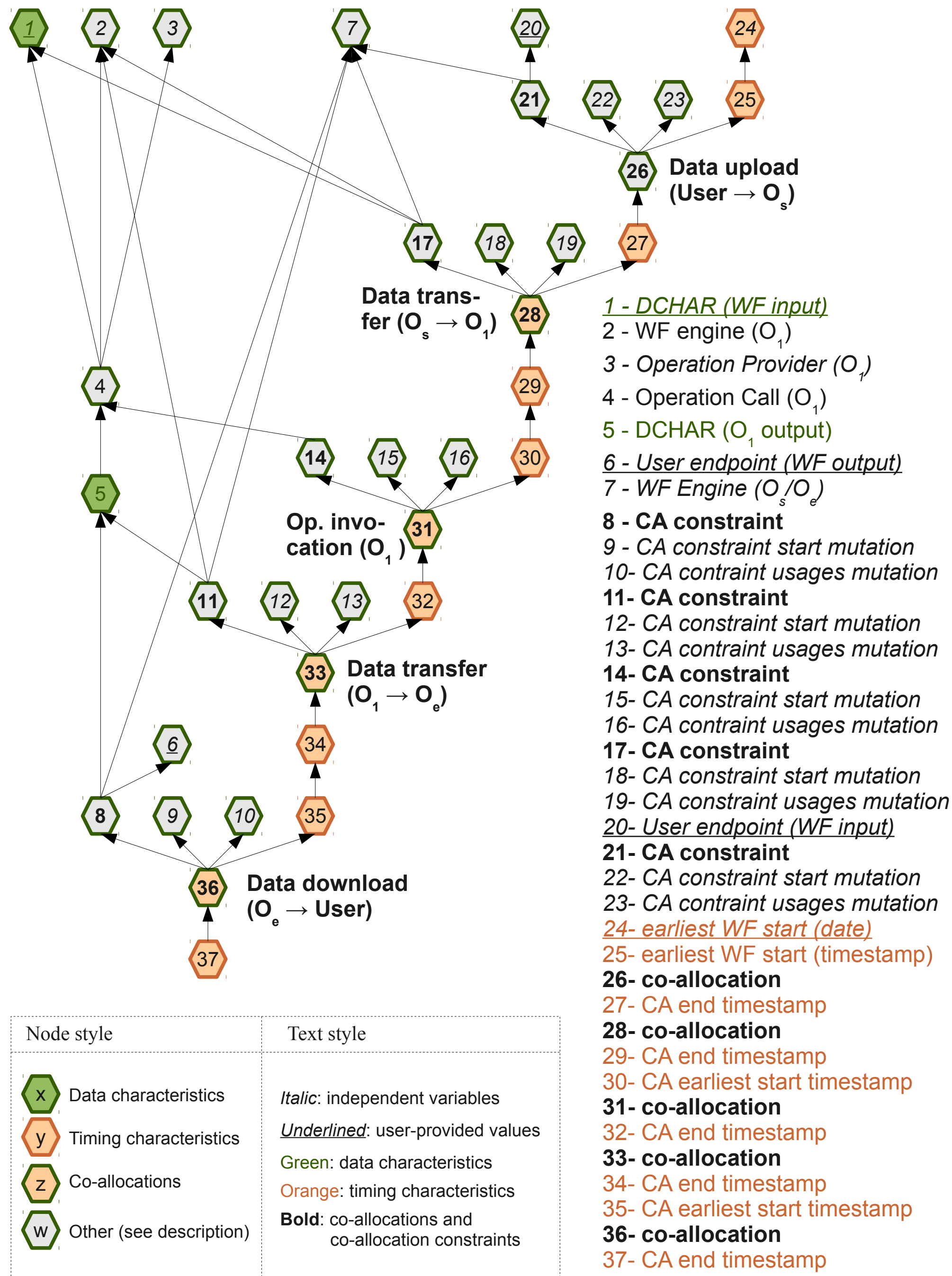


Figure 3.7: Variable Dependency Graph of a simple Workflow

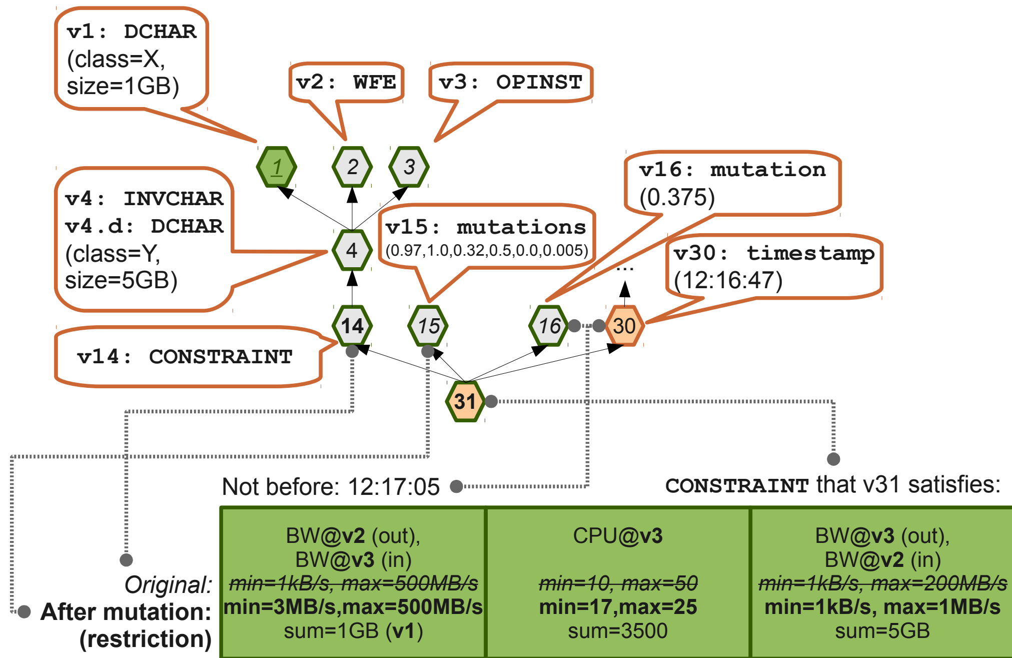


Figure 3.8: Annotated Variable Dependency Graph Fragment

2. **Chromosome**: The structure of the chromosome is directly derived from the workflow description. The chromosome is aware of which workflow steps have infrastructural requirements (e.g., need an operation instance or a workflow engine) and is able to change their assignments via mutating the corresponding genes; it also "knows" how co-allocation constraints are correctly determined and restricted (via mutation). Internally, all values and calculations are represented as variables; dependencies are expressed as variable dependencies within or across the genes.

3. **Variable Dependency Graph**: The VDG is built at runtime from the chromosome. At this level of abstraction, we are only considering dependencies between individual variables without needing to know about their semantics – in other words, if the VDG is constructed correctly, during its evaluation it is not necessary to know which particular allele, or calculation, of which particular workflow step a variable represents.

4. **Workflow Schedule**: The variable parts of the workflow schedule can all be determined from the Chromosome: the workflow engine mapping Ξ and Operation instance mapping Ω are directly contained within alleles, the co-allocations C in turn are calculated by the VDG. Actually, C is simply the union of the values of all co-allocation values within the VDG.

3.3.5 Mutations

Mutation is one of the strategies employed by a GA to generate new individuals, i.e., during the breeding phase of a new generation. Logically, the new individual is origi-

nally a copy of its ancestor, but then experiences modifications of some of its allele values. When considering the individuals at the level of the Variable Dependency Graph, the graph (and the values of the variables) are initially identical. Mutations affect individual alleles, and changes must be propagated to dependent variables – thus possibly triggering a cascade of recalculations. In order to perform only the required recalculations, we leverage the fact that the variable dependency graph is topologically sorted. Intuitively, updates are performed in a breadth-first manner in the graph, which allows to i) perform recalculations in the correct order (preserving dependencies), and ii) avoid recalculating a single variable's value multiple times. Pseudocode for the recalculation is given in Algorithm 2.

Algorithm 2: Updating Variable Dependency Graph after Mutations

```
Input: an array D of variables whose alleles have been mutated, sorted by the
       variables' identifiers in ascending order
// D is mutable; its values are kept sorted at all times
while D is not empty do
    Variable v ⇐ D[0];
    recalculate v's value;
    if v's value has changed then
        foreach Variable d: d depends on v do
            if D does not contain d then
                insert d into D;
    remove D[0];
```

3.3.6 Determining Compliant Co-Allocations

The computationally most expensive task is to find valid co-allocations. To briefly recap the goal that must be achieved: given a co-allocation constraint C, a timestamp t, a reservation state S, and a planning timestamp p, find a co-allocation K such that

$$K \xrightarrow{\checkmark} C \wedge \text{coAllocStart}(K) \geqslant t \wedge \text{feasible}(S, p, \{K\}) = \texttt{true}$$

As there is an infinite number of co-allocations satisfying these properties, we employ what could be called a greedy strategy, by further restricting the goal to find the matching co-allocation which starts earliest and has the shortest duration.

Before going into further detail, let us consider for a moment the context in which this search is performed: co-allocations are needed for any workflow step which implies resource usage, such as operation invocations or data transfers. However, these workflow steps may be executed in parallel with other workflow steps, which might require co-allocations for the same resources at the same time. When these searches are performed independently (resulting for instance in co-allocations K_1 and K_2), considering only the reservation state S, one might end up "overbooking" some resources, i.e., $\text{feasible}(S, p, \{K_1\}) = \texttt{true} \wedge \text{feasible}(S, p, \{K_1\}) = \texttt{true}$, but $\text{feasible}(S, p, \{K_1, K_2\}) = \texttt{false}$.

Dealing with parallelism

Clearly, in addition to the reservation state S, one must also consider some – but not all – of the (not yet reserved) co-allocations of other workflow steps. Since all co-allocations are calculated within Variable evaluations, one can reason on the variable dependency graph. Figure 3.9 shows a fragment of a very simplified variable dependency graph. Note that i) this figure serves as a condensed example of what parallel patterns may look like, and is not related to the previous figures; ii) only the most important variables relevant to co-allocation search have been depicted, other variables have been omitted; iii) the numbering of the variables was done manually – the actual implementation would result in a different one. However, the reasoning below holds for *any* topological sort, and is easier to follow with the numbering presented in the figure.

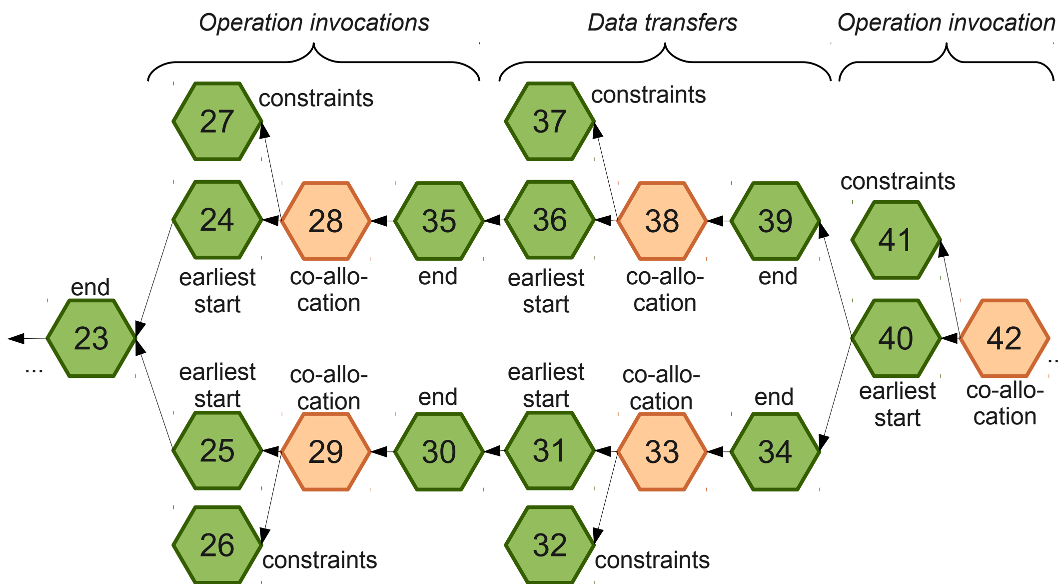


Figure 3.9: Simplified Variable Dependency Graph Fragment with Parallelism

The variables which “produce” co-allocations (termed co-allocation variables hereafter) are the ones of interest, and have been highlighted in orange in the figure. Because the scheduling must observe temporal dependencies, co-allocation variables always depend on a variable which indicates the earliest start, and always have a dependent variable indicating their end. Put differently: if a co-allocation variable S (transitively) depends on a co-allocation variable P, then it is impossible for the co-allocation resulting from S to start before P has finished.

Let us now come back to the planning itself. Once an individual (a “chromosome instance”) has been evaluated, all its variable values have been calculated. In particular, all co-allocations have been assigned such that the aggregation of all of them represents a valid schedule for the workflow. Suppose that this individual has been chosen for breeding. Cloning it results in a new individual, whose variable values are initialized with the cached values from its predecessor – in other words, it represents the same schedule. Now suppose that a mutation is performed on variable 27 in Figure 3.9, thus changing the co-allocation constraints, and invalidating co-allocation variable 28. In

order to calculate it afresh, which co-allocations (both from the reservation state, and the individual itself) do we need to take into account, and which should we ignore? There is an obvious trivial case: we must ignore the co-allocation that variable 28 is currently holding, because it is the one we are overwriting. More generally, there are five rules to consider when evaluating a co-allocation variable V:

1. Consider all allocations imposed by the reservation state of the system. These allocations have been reserved and must be obeyed. We will refer to these allocations as *committed* allocations.

2. Ignore the allocations currently held by V, as these allocations are replaced by the re-calculation.

3. Ignore allocations held by co-allocation variables which V (transitively) depends on. There cannot be any temporal overlap between these allocations and V, because V's allocation must only start after all of the dependees' allocations have finished.

4. Ignore allocations currently held by co-allocation variables which (transitively) depend on V.

5. Consider all allocations held by all other co-allocation variables of the variable dependency graph. We refer to these as *uncommitted* allocations.

While the first three items are intuitively clear, we will now briefly focus on the latter two. Consider what happens *after* V's recalculation is finished: regardless of the actual new co-allocation, we can limit ourselves to only observe the timing behaviour. If the end timestamp has not changed, then no further action is necessary. If it has indeed changed, a (possibly cascading) re-calculation of dependers is performed: any co-allocation variable which depends on V will be recalculated, thus correcting any possible overbooking. In fact, it is even *necessary* to ignore dependers' allocations, as otherwise one might incorrectly "miss" available resources. Figure 3.10 shows a trivial example illustrating both cases.

While V is being re-calculated, allocations associated with V, with V's (transitive) dependees, and with V's (transitive) dependers should thus be ignored. All co-allocation variables which are not in these categories necessarily neither depend on, nor are depended on by, V. In other words, they may be in parallel to V, and thus must be considered. Ignoring them might result in the case outlined above, namely in the overbooking of resources.

Unfortunately, given only the topologically sorted IDs of variables, it is not practically possible to determine which variables depend on which others from the IDs alone. It is therefore necessary to maintain a precedence matrix to keep track of these relations.

Allocation Representation

As introduced in Section 2, an allocation is merely a set of contiguous usage blocks, and a usage block consists of start and end timestamps and the actual usage amount.

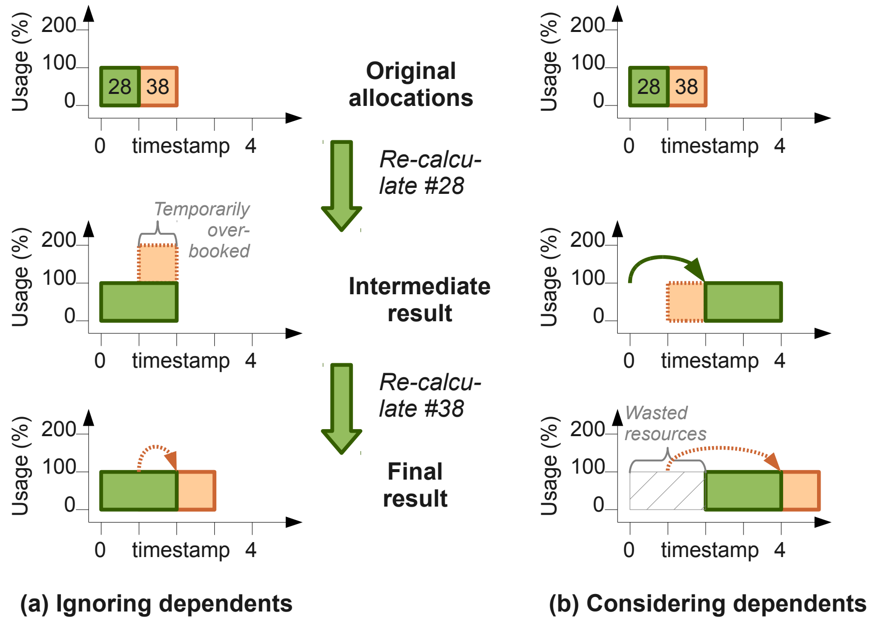


Figure 3.10: Strategies for dependent Co-allocation Variables during Mutations

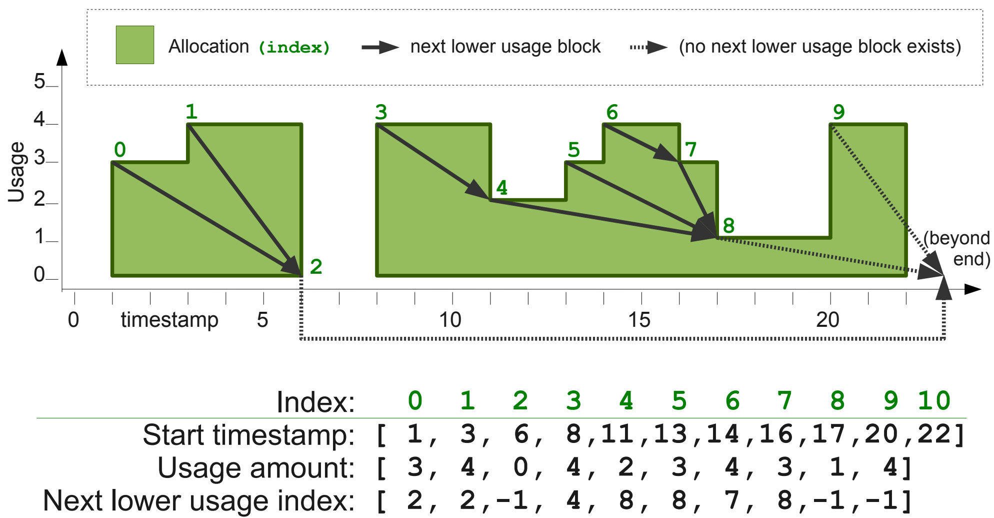


Figure 3.11: Graphical and internal Representation of an Allocation

Because of these properties, allocations can be represented in a very compact manner, by using arrays of simple types.

Figure 3.11 shows an example of an allocation, and how it is represented internally. In addition to the two arrays storing the start timestamps and usage amounts of the individual usage blocks, we have introduced a third array, which, for each usage block, contains a forward pointer to (the index of) the first subsequent usage block with a strictly lower usage amount. The complete array is cheap to construct (it can be initialized in linear time with regard to the number of usage blocks), but may speed up the finding of (new) co-allocations: the sum of existing and new allocations must never use a resource beyond its capacity. If the new allocation is not feasible "on top" of the existing ones, then these pointers allow to speed up the determination of the next timestamp where the search can be retried. As statistically, there is a 50/50 chance of the next usage block being higher or lower, the average speedup is by a factor of two.

Note that the allocation shown is indeed a single allocation, not two independent ones. The definition of allocation explicitly allows for usage blocks with zero usage. Such allocations with "gaps" are often encountered when adding individual ("gapless") allocations which do not overlap. As described below, we perform such additions to aggregate all *committed* allocations into a single one during search.

Finding Co-Allocations

Consider again Figure 2.6 (p. 19). It graphically depicts how a co-allocation constraint may look like. In simple terms, we must satisfy a sequence of time-determining allocation constraints, indicating which resources must provide some specified total (sum), while only utilizing a certain range of the resources' capacities (min and max). If a time-determining allocation constraint refers to more than one resource, the exact same allocations (in terms of usage blocks) must be feasible on all resources concurrently. Moreover, there may be an arbitrary number of dependent allocation constraints, referring to one resource each, indicating a minimum resource usage to be allocated, and the time-determining allocation constraints that the dependent allocation constraint must start and end with (i.e., the ones it must "run with" in parallel).

The goal thus is to find a set of allocations which satisfies all those allocation constraints. As previously explained, we know the timestamp t at which we must start looking for these allocations, and we can determine all other (pre-existing, committed or uncommitted) allocations that we must consider while searching. Because they are not changing during the planning, we internally keep all *committed* allocations for the same resource aggregated (summed up) as a single allocation – intuitively, this roughly corresponds to the concept of *materialized view* in database systems.

At runtime, we consider all allocations of all required resources at the same time. A very simplified and high-level pseudocode representation of the implementation is shown in Algorithm 3, and briefly discussed below.

- **Lines 2, 3, 15, 18, 19**: *currentStart* is the variable that contains the timestamp where the entire (prospected) co-allocation starts, while *currentTs* is the timestamp that is currently considered while "building" the co-allocation.

Algorithm 3: Finding compliant Co-allocations

```
Input: a co-allocation constraint C, a timestamp t, and a set A of allocations to be
       considered
Output: a co-allocation K such that K ✓→ C
1  K ⇐ ∅;
2  currentStart ⇐ t;
3  currentTs ⇐ currentStart;
4  while true do
5      TAC TC ⇐ the next unfulfilled time-determining allocation constraint ∈ C;
6      if TC = NIL then break;
7      DAC DC[] ⇐ dependent allocation constraints ∈ C which are parallel to TC;
8      ALLOC TA ⇐ ∅;
9      ALLOC DA[] ⇐ [∅, . . . , ∅];
10     while ¬(TA ✓→ TC) do
11         ALLOC W[] ⇐ {a ∈ A| allocEnd(a) ⩾ currentTs ∧ a.r ∈ {RESOURCE r|r ∈
           TC.r ∨ r ∈ DC[i].r}};
12         nextEvent ⇐ the lowest timestamp > currentTs where any allocation a ∈ W
           starts a new usage block;
13         if given W, during [currentTs, nextEvent[, every constraint in DC can be
           fulfilled and remaining capacity ∀r ∈ TC.r is in [TC.min, TC.max] then
14             create and add corresponding usage blocks to TA and each DA[i];
15             currentTs ⇐ nextEvent;
16         else
17             nextTry ⇐ the lowest timestamp where any allocation of a saturated
               resource exhibits lower usage;
18             currentStart ⇐ currentStart + (nextTry − currentTs);
19             currentTs ⇐ currentStart;
20             mark all time-determining allocation constraints as unfulfilled;
21             K ⇐ ∅;
22             continue outer loop;
23     K ⇐ K ∪ {TA} ∪ DA;
24     mark TC as fulfilled;
25 return K
```

- **Outer loop (lines 3–24)**: This fulfills the time-determining allocation constraints, one at a time, in the order that they are given in the co-allocation constraint. For every time-determining allocation constraint, its dependent allocation constraints are also considered. In other words, at the end of the loop (**line 24**), the following will hold: $TA \xrightarrow{\checkmark} TC \wedge DA[i] \xrightarrow{\checkmark} DC[i]$.

- **Inner loop (lines 10–22)**: This loop is repeated as long as the currently considered time-determining allocation constraint is not fulfilled.

- **Lines 11–12**: All currently relevant allocations (pertaining to any allocation constraint to be fulfilled, and not having ended before *currentTs*) are determined. The next event (timestamp) to consider is when any of these allocations changes its usage amount.

- **Lines 13–15**: If all constraints are satisfiable, then the prospected allocations are extended throughout the next event timestamp. Eventually, this will also completely fulfill the current time-determining allocation constraint (*TC.sum* is reached), thus ending the inner loop.

- **Lines 17–22**: If any constraint was not satisfiable, the entire co-allocation is not feasible starting at the given timestamp (*currentStart*), and we must backtrack. The entire co-allocation is reset, and *currentStart* is shifted into the future so as to avoid the currently encountered "bottleneck". Finally, the search is started over.

As said, this description is simplified to present the idea of the implementation, and omits a few details. The actual implementation is more sophisticated – for example, the set of allocations to consider, and the events (timestamps where new usage blocks begin) are organized in priority queues, which avoids unnecessary recalculations.

The actual runtime of the algorithm (and the number of required backtracking steps) is largely dependent on pre-existing allocations and the co-allocation constraint to be satisfied. However, the algorithm is guaranteed to eventually terminate: in the worst case, all required allocations would be made after all existing allocations.

3.4 Fitness Functions in DWARFS

The previous section presented details about the structure and interpretation of the chromosome employed to represent workflow schedules. While this is an important asset for a genetic algorithm, it is not sufficient – equally important is the definition of an appropriate fitness function.

The fitness function that one chooses for a GA optimization is of crucial importance, as it represents both the optimization goal, and the method for assessing the suitability of individual solutions with respect to that goal. An important property of fitness functions is that they assign a single numerical value, namely its fitness, to any potential solution – i.e., any schedule in the DWARFS case. These individual fitnesses must be comparable in order to determine which candidate solution is better. Ideally, such a comparison between the fitnesses of two individuals would also quantify the degree of preference (which we call preferability), thus allowing to say, for example, "schedule *A* is 3 times better than schedule *B*".

In the following, we describe a few characteristics of the metrics that we defined for schedules, and of the optimization objectives that we wish to support, and discuss how they relate to the properties of fitness functions.

3.4.1 Semantics and Comparability of Individual Schedule Metrics

All of the metrics that we consider (duration, termination time, cost) have a "natural" associated optimization goal, namely to minimize the respective value. In other words, "as cheap as possible", "as fast as possible", "finish as early as possible" present reasonable objectives, while the opposite (e.g., "run the process as expensive as possible") simply does not make sense (though it could be supported by the system). So while we can clearly say that lower values are preferrable to higher ones, can we also quantify how much preferrable they are?

A simple approach is to directly map the absolute value of any metric to a fitness value (in the case where the goal is to minimize the respective metric, one needs to additionally change the fitness evaluation to "smaller is better" semantics, or to slightly transform the absolute value, e.g., by either negating or inversing it). While this clearly allows to order candidate solutions with regard to their fitness, it will not allow to express preferability.

This is easiest demonstrated by using an example: consider an optimization for cost only. If a population consists of three candidate schedules s_1, s_2, s_3, with their associated cost being 10, 20, and 50 dollars respectively, is it permissible to say that s_1 is 5 times "better" than s_3? What would the preferability ratios be if the costs were 1010, 1020, and 1050 dollars? The problem is that while one knows the absolute values, one cannot normally quantify how "good" each of these values really is. In other words, 1010 dollars might be a close-to-optimal value for a complicated and resource-demanding workflow, but outrageously expensive for a small and simple one; the chicken-and-egg issue is that one cannot *absolutely* quantify preferability unless an absolute ground truth (the optimum) is known – which is precisely the value that is not known beforehand.

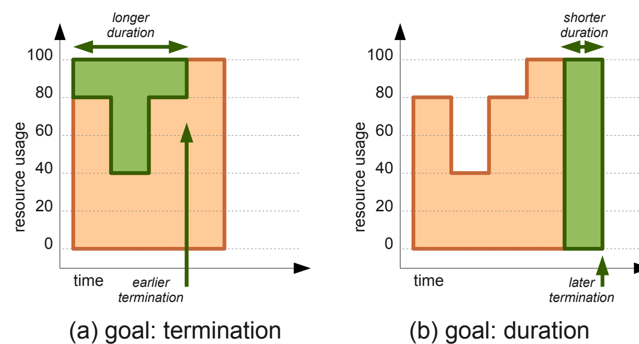
3.4.2 Multi-objective Optimization

One further complication concerning fitness functions and fitnesses is that a fitness function produces a single numerical value, but DWARFS is also meant to support multiple optimization objectives.

The metrics that we introduced can be regarded as multiple dimensions: *termination*, *time*, and *cost*. The metrics of a single schedule along these dimensions are not normally orthogonal, but somewhat interrelated. Unfortunately, it is not generally possible to predict *how* exactly they relate to each other.

For example, *cost* and *duration* generally are negatively correlated: higher resource usage costs more, but results in faster execution. However, this is not necessarily true in all cases (e.g., co-locating services may use a lot of bandwidth at a single cloud provider, but for free).

Similarly, *duration* and *termination* are generally positively correlated. In a system without Advance Reservations, or in a completely empty infrastructure, they even completely collapse into the same goal (correlation 1.0). On the other hand, for a rather loaded, but not saturated, infrastructure, the opposite may become true. A trivial example (where a single resource is affected) is shown in Figure 3.12.



(a) goal: termination (b) goal: duration

Figure 3.12: Planning Goals: Duration vs. Termination in Loaded Infrastructure

In general, we thus cannot know whether or how a score on one dimension influences a score on another dimension. Multiple optimization goals may contradict each other in one infrastructure state, while they may be fullfillable at the same time in a different configuration.

The possibility to specify multiple optimization goals, especially if they are contradicting, leads to another issue: for instance, when optimizing for both cost and duration, a candidate schedule s_1 which is cheap and long-running, and an expensive and short-runing schedule s_2 are not comparable anymore: each is better than the other in one dimension, but which is the one that is preferrable overall? One approach is to consider the set of pareto-efficient individuals by considering all dimensions independently. In fact, this corresponds to establishing a partial order over the population. However, this clashes with the property of a fitness function, namely that it must establish a total order over all candidates.

A simple and widely practiced approach, which we also adopted in DWARFS, is using weighted sums of the scores (fitnesses) on the individual dimensions. However, this leads to one more challenge, which is again related to the abovementioned issue of the semantics of absolute values of the individual metrics. More precisely: not only do we now have multiple dimensions, where the absolute values among each dimension are impossible to quantify in terms of preferability, but we also need to combine (by means of weighted sums) values from entirely different domains.

3.4.3 Normalized Fitness Functions

To overcome both of the abovementioned issues, we have devised normalized fitness functions. These fitness functions allow to assign normalized fitness values to individuals of a population, regardless of the metric being considered (i.e., regardless of the domain that the metric represents, and of the absolute values that are present).

Before going into detail about the concept and definition, let us again stress the most important properties of a fitness function. First and foremost, it must assign higher values to better individuals (according to the optimization goal that the fitness function represents). Ideally, the fitnesses of individuals should also be proportional to their preferability. As discussed above, there is no *absolute* ground truth – in other words, absolute bounds – allowing to establish such proportions. There are, however, *relative*

bounds: because fitness functions are always used within the context of a population of individuals, one can use the minimum and maximum values of the range encompassed by the individuals as bounds. In other words: a normalized fitness function determines the fitness of an individual in relation to the population it is part of.

More formally, a normalized fitness function observes the following properties:

- The fitness of every individual of a population is directly proportional to its preferability within that population.
- The sum of the fitnesses of all individuals of a population is 1.

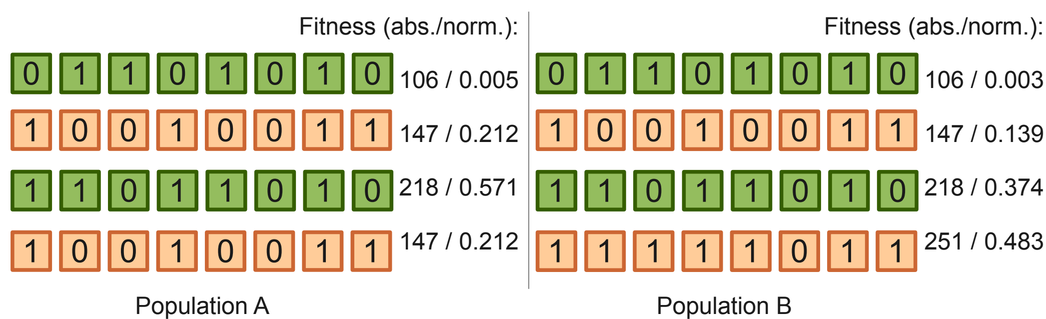


Figure 3.13: Normalized Fitness Function

There are three properties immediately following from this specification: First, an individual is no longer always assigned the same fitness; rather, each individual's fitness depends on all other individuals in the population. Second, the fitness is completely disassociated from the values of the domain it represents – however, both the ordering and the proportions between individuals are preserved. Third, multiple normalized fitness functions can be combined into a new (normalized) fitness function by leveraging simple weighted sums. Note that this last statement slightly "waters down" the term *fitness function*: in the strict sense as used in the GA literature, there is a single fitness function which applies to the *entire* optimization goal. From here on, we will also use the term to refer to the determination of normalized fitness values in a single dimension.

Figure 3.13 shows the effect of using a normalized fitness function: both populations differ only in a a single individual. However, because the normalized fitness value is relative to the entire population, individuals with the same absolute metric value get assigned different fitness values in both populations.

Normalization procedure

Let us consider metrics which have larger-is-better semantics first. Given a population (or sample) of individuals, the simplest possible normalization is to assign each individual its absolute metric value, divided by the sums of the absolute values of all individuals. We refer to this approach as `REL-ASC-MIN`. Except in the pathological case where all absolute values are 0 (which needs to be handled separately) , this results in valid fitness values. However, it still leaves the problem of interpreting preferability: in

Algorithm 4: Normalizing Fitness Functions Pseudocode

```
Input: an array A of absolute values of some metric, each element representing
       the value of one individual in the population
Output: an array F of normalized fitness values for the population
N ⇐ A;
min ⇐ min_{v∈N} v;
max ⇐ max_{v∈N} v;
sum ⇐ 0;
foreach v ∈ N do
    if the values of A have smaller-is-better semantics then
        v ⇐ max − v;
    else
        v ⇐ v − min;
    if shifting should occur wrt. the minimal value then
        v ⇐ v + min;
    else if shifting should occur to avoid 0 fitnesses then
        v ⇐ v + 1;
    sum ⇐ sum + v;
if sum = 0 then sum ⇐ |N|;
foreach v ∈ N do
    add v/sum to F;
return F
```

a population where all individuals span a small range of large values (i.e., when minimum and maximum values of the sample are large, but close), the absolute differences tend to disappear, thus assigning almost equal fitnesses to all individuals.

One approach is to *shift* the entire population in order to emphasize on the differences – for example by subtracting the overall minimum from all individuals first. This approach, termed `REL-ASC-0`, tends to capture the differences between individuals better, regardless of the actual values. It however produces a new problem, namely that the least suitable individual (the one with minimal value), gets assigned a fitness of 0. This may not be an issue in all cases, but for instance, it would completely preclude that individual from breeding in a roulette wheel selection setup.

A possible solution to that problem is to simply add a constant (for example 1), to all shifted values. This is somewhat of a pragmatic approach – it guarantees that no individual is left with a fitness value of 0. Still, the value of the least suitable individual is determined by the range of values encompassed by the entire population ($\frac{1}{max-min}$). We call this approach `REL-ASC-1`.

When a metric has smaller-is-better semantics, the approaches are very similar to the abovementioned, and we denote them accordingly as `REL-DSC-MIN`, `REL-DSC-0`, `REL-DSC-1`. In fact, Algorithm 4 presents pseudocode which handles all of these cases.

Table 3.2 presents details on how various normalization approaches operate on different input. We have specifically chosen the three populations $< 10, 20, 50, 100 >$, $< 1010, 1020, 1050, 1100 >$, and $< 100010, 100020, 100050, 100100 >$ because of their

Fitness function	Input data:	<10, 20, 50, 100>	<1010, 1020, 1050, 1100>	<100010, 100020, 100050, 100100>
ABS-ASC	fitnesses	**<10, 20, 50, 100>**	**<1010, 1020, 1050, 1100>**	**<100010, 100020, 100050, 100100>**
REL-ASC-MIN	normalized data	<10, 20, 50, 100>	<1010, 1020, 1050, 1100>	<100010, 100020, 100050, 100100>
	fitnesses	**<0.056, 0.111, 0.278, 0.556>**	**<0.242, 0.244, 0.251, 0.263>**	**<0.250, 0.250, 0.250, 0.250>**
REL-ASC-0	normalized data	<0, 10, 40, 90>	<0, 10, 40, 90>	<0, 10, 40, 90>
	fitnesses	**<0.000, 0.071, 0.286, 0.643>**	**<0.000, 0.071, 0.286, 0.643>**	**<0.000, 0.071, 0.286, 0.643>**
REL-ASC-1	normalized data	<1, 11, 41, 91>	<1, 11, 41, 91>	<1, 11, 41, 91>
	fitnesses	**<0.007, 0.076, 0.285, 0.632>**	**<0.007, 0.076, 0.285, 0.632>**	**<0.007, 0.076, 0.285, 0.632>**
REL-DSC-MIN	normalized data	<100, 90, 60, 10>	<1100, 1090, 1060, 1010>	<100100, 100090, 100060, 100010>
	fitnesses	**<0.385, 0.346, 0.231, 0.038>**	**<0.258, 0.256, 0.249, 0.237>**	**<0.250, 0.250, 0.250, 0.250>**
REL-DSC-0	normalized data	<90, 80, 50, 0>	<90, 80, 50, 0>	<90, 80, 50, 0>
	fitnesses	**<0.409, 0.364, 0.227, 0.000>**	**<0.409, 0.364, 0.227, 0.000>**	**<0.409, 0.364, 0.227, 0.000>**
REL-DSC-1	normalized data	<91, 81, 51, 1>	<91, 81, 51, 1>	<91, 81, 51, 1>
	fitnesses	**<0.406, 0.362, 0.228, 0.004>**	**<0.406, 0.362, 0.228, 0.004>**	**<0.406, 0.362, 0.228, 0.004>**

Table 3.2: Sample Results for Various Fitness Determination Strategies

characteristics: the individuals of all populations are all spread in the same way over an interval of the same size, but at different absolute "offsets".

To ease the understanding, we have included the normalized absolute values in addition to the final fitnesses (i.e., the values of N and F of Algorithm 4, respectively). The examples clearly show that:

- the absolute-value fitness function `ABS-ASC`, and the non-shifting normalized functions `REL-ASC-MIN` and `REL-DSC-MIN` render the individuals almost indistinguishable at high absolute values – i.e., the higher the absolute values are, the more small differences are "filtered out".

- conversely, normalized functions which do employ shifting consider only the relative range of values, by ignoring the absolute values. In the wording we used above, these function establish an absolute ground truth using the extrema of the population itself. This ultimately results in the final fitness values being the same for all three populations.

Comparison

In order to evaluate the applicability and performance of normalized fitness functions, we have performed a number of tests using a simple optimization objective. The goal was to find the largest and smallest unsigned 63-bit integer, respectively. In all cases, we used a population size of 16 individuals, with weighted roulette selection and elitism (carrying over the single best individual of each generation to the next).

The results are shown in Figure 3.14, and are briefly discussed below. For each fitness function, 1000 optimization runs over 100 generations were performed; the graphs show how the best individual at each generation performed. The results show, at least for the simple optimization task chosen, that all normalizations lead to effective results with more or less similar convergence behavior. More importantly though, because of their properties, the values resulting from normalized fitness functions are directly suitable for further processing, such as the weighted sums that we use for multi-objective optimizations.

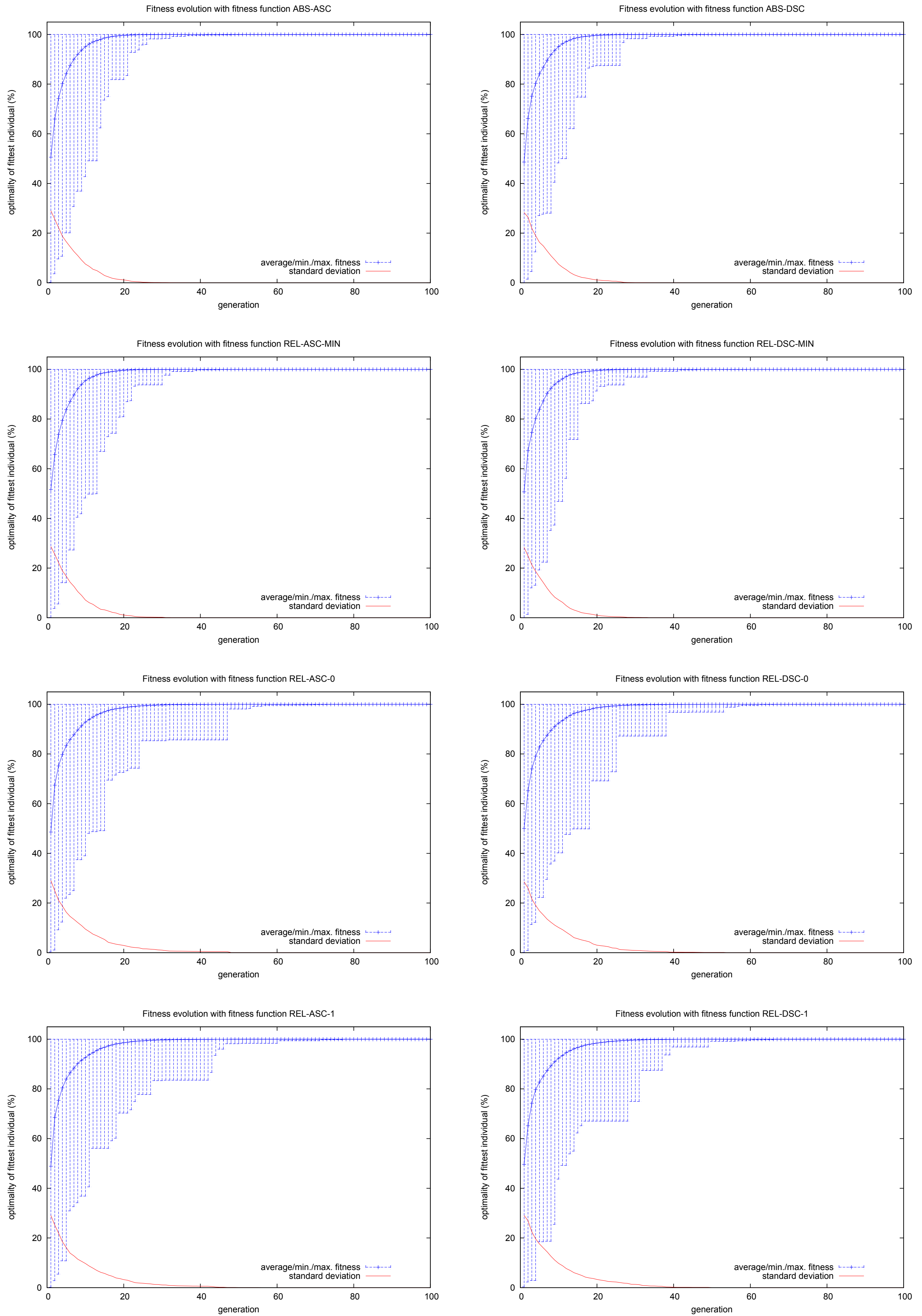


Figure 3.14: Comparison of various Fitness Functions

3.4.4 Bounds

We have previously described how fitness functions can be normalized, and ultimately be combined to allow for multi-objective optimization. There is another aspect of the user requirements that we have not discussed yet, namely the ability to specify bounds for selected metrics. For instance, users could specify an objective such as "terminate as early as possible, but do not exceed a budget of 20 $". While the evaluation of bounds is a trivial binary decision, there is one minor issue: if a population does not contain any individual satisfying such a bound, which strategy should be employed to find such individuals? The answer is simple and intuitive: one (temporarily) switches the optimization goal to satisfy the bound – in this case, optimizing for cost – until the bound is reached. For this reason, bounds are also associated with fitness functions which drive the optimization to reach the bound.

3.4.5 Fitness Evaluation

In Section 3.1 we had stated that as the most general case, the DWARFS planner supports multi-objective optimization with multiple bounds. The evaluation of the fitnesses of a population's individuals is in fact performed as a three-step process, which we outline in the following.

Filtering

We support an arbitrary number of bounds on different metrics, such as deadline or budget constraints. These are usually, but not necessarily, objectives that the overall fitness function optimizes *against* – e.g., optimizing for fast execution normally drives costs up.

All individuals are subject to evaluation of all such constraints *before* the actual fitness determination. If an individual violates any of the constraints, it is assigned a fitness of 0 and removed from further considerations – i.e., bounds act as filters on the population, by only leaving in compliant individuals. It may happen that a filter would remove *all* individuals because they all violate its constraint, especially if the constraint is tight, or if the optimization is still in an early stage. In this case, we temporarily ignore subsequent filters, and replace the fitness function by the one that is associated with the bound. Intuitively, this corresponds to first optimizing for the filter goals, so as to obtain some valid individuals, and only then going for the actual objective.

Fitness Evaluation

In the second step of the fitness evaluation, we calculate the fitnesses of all individuals that have been determined to be valid by the filters. For single-objective optimizations, the fitness function does not necessarily need to be (but may be) normalized; multi-objective optimizations are performed by using weighted sums of multiple normalized fitness functions.

Ordering and Selection

The final step in the fitness evaluation is an additional ordering of the individuals, best motivated by an example: When optimizing for duration only, several plans may take the exact same time, but start (and end) at significantly different timestamps and have different costs. When considering *only* the final fitness values this distinction is lost, and any of these individuals, at random, could be considered the best. This final step thus allows to apply secondary preferences to rank individuals with the same fitness.

3.5 Partitioning and Data Transfers

The previous sections presented details about the "inner workings" of the GA approach that we employ for planning a workflow execution in DWARFS: in simple terms, chromosomes are built from the workflow definition, their alleles influence the timing and resource usage behavior, and ultimately each chromosome defines a concrete workflow schedule, in terms of the resulting co-allocations. Fitness functions are used to determine the suitability of the chromosome with respect to the QoS criteria, whereby the chromosome with the best overall fitness represents the best (currently known) schedule with respect to user criteria.

At the lowest level, such a workflow schedule is fully determined by its contained co-allocations. However, if we take one step back and take a look at the schedule as a whole, we can observe a few more interesting properties, depicted in Figure 3.15, and discussed in the following sections.

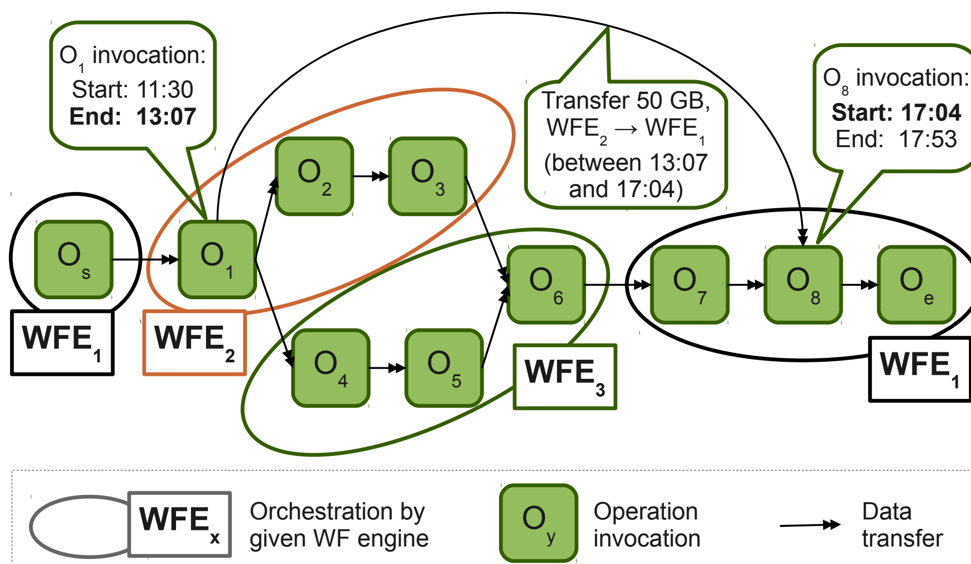


Figure 3.15: Example of Process Fragmentation and Data Transfers

3.5.1 Partitioning

DWARFS was designed with distributed execution in mind, where the execution of the workflow is carried out by (possibly) multiple co-operating workflow engines. In the most extreme case of distribution, each activity might be handled by a different workflow engine. In practice however, we usually observe a kind of clustering – or execution partitioning – , where the same workflow engine handles multiple operation invocations (the other extreme, in terms of clustering, would be a single engine carrying out the execution of the entire workflow). This partitioning could be considered a (wanted) side-effect of the optimization, and depends both on the optimization goals, and the infrastructure state.

Why such partitioning occurs and is desirable is intuitively clear when considering the properties of the infrastructure: if instances of two operations, representing consecutive activities in the workflow definition, are available in the same network segment, it is often preferrable to execute both of them using a single workflow engine which is in proximity to both operation instances. Network transfers for the operation invocations are bound to be fast, and no additional transfers are required to hand over control between workflow engines. Furthermore, network transfers may also be cheaper (or be completely free) if all transfers happen within the infrastructure of the same cloud provider.

However, the abovementioned statement does not necessarily hold in all conditions: There simply may not exist co-located operation instances or workflow engines, or existing reservations may dictate that another instance assignment fulfills the optimization goals better. The important property of the DWARFS planner is that it is adaptive, in the sense that the best partitioning scheme may vary depending on the current reservations, and the user's QoS requirements.

3.5.2 Data Transfers

Another interesting aspect, also shown in Figure 3.15, stems from the combination of a few properties of the system: first, we are interested in long-running Scientific Workflows; second, the execution is distributed; third, we establish reservations for resources for concrete time frames. The combination of these properties may lead to situations where data that is produced early in the workflow execution, by an operation handled by some workflow engine, is only required substantially later, for an operation handled by a different workflow engine. While the data will need to be shipped from the source to the destination, this transfer does not necessarily have to be performed immediately, nor at full speed.

Figure 3.16 depicts the resources used for data transfers between (possibly transitively) succeeding workflow activities, handled by different workflow engines. The most important aspect to note is that a workflow engine becomes active for an activity as soon as the first incoming data transfer for that activity starts coming in; conversely, it must stay active until the last outgoing transfer is completed. In practice, this means that if only direct data transfers are allowed, data which is produced early, but required late, would unnecessarily occupy resources at workflow engines.

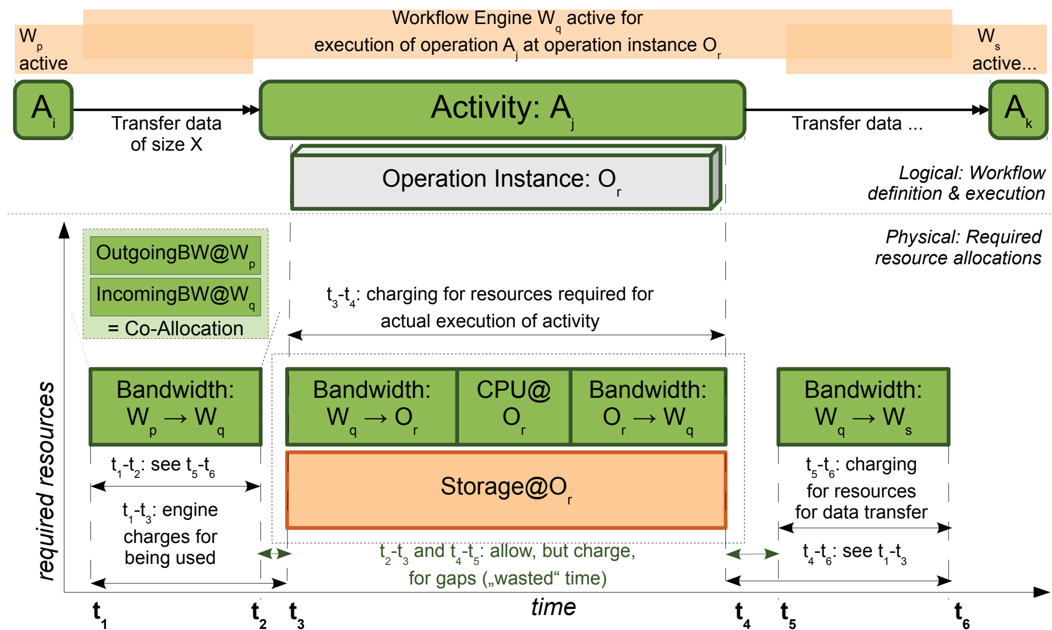


Figure 3.16: Resource Usage for Inter-Engine Data Transfers

Storage Nodes and Transfer Strategies

For this reason, we have introduced new entities termed Storage Nodes (SN). Storage nodes are meant to provide reliable and cost-effective intermediate storage capacity. One example where storage nodes are beneficial has already been discussed above, however there are also other cases in which more intricate data transfer strategies may be beneficial.

We have identified four useful strategies for data transfers:

1. Direct transfer: transfers output data directly to the workflow engine that needs them as input. This strategy is normally preferrable for directly succeeding activities, or generally when the data transmission takes longer than activities that run in parallel to the transfer.

2. Indirect transfer: output data is transferred from the source to a storage node first, where it is stored for some time. It is afterwards transferred to its destination, ideally "just in time" before it is needed for execution. This strategy is normally preferrable for data which is produced early, but required late – in other words, when there are other long-running activities; we thus avoid shipping data to a workflow engine much ahead of time.

3. Double-indirect transfer: this is a variation of the indirect transfer which uses two (consecutive) intermediate storage nodes instead of one. The rationale is that when the source workflow engine and the target workflow engine are located in different, poorly-connected networks, it may be preferrable to quickly save the output to a "close" (well-connected) storage node, then use the available time to transfer the data, and finally quickly retrieve them.

4. Triple-indirect transfer: This strategy adds another level of indirection to the double-indirect transfer, for cases where a direct connection between the source Storage Node and the target Storage Node may be slower than "routing" the data through a third party.

In principle, indirect data transfers with even more "hops" can be supported by the system; however, we do not see the benefit of allowing for such kind of transfers at the moment. The alternative strategies are depicted in Figure 3.17. We will come back to this figure, and more thoroughly discuss the resource requirements, in a short while.

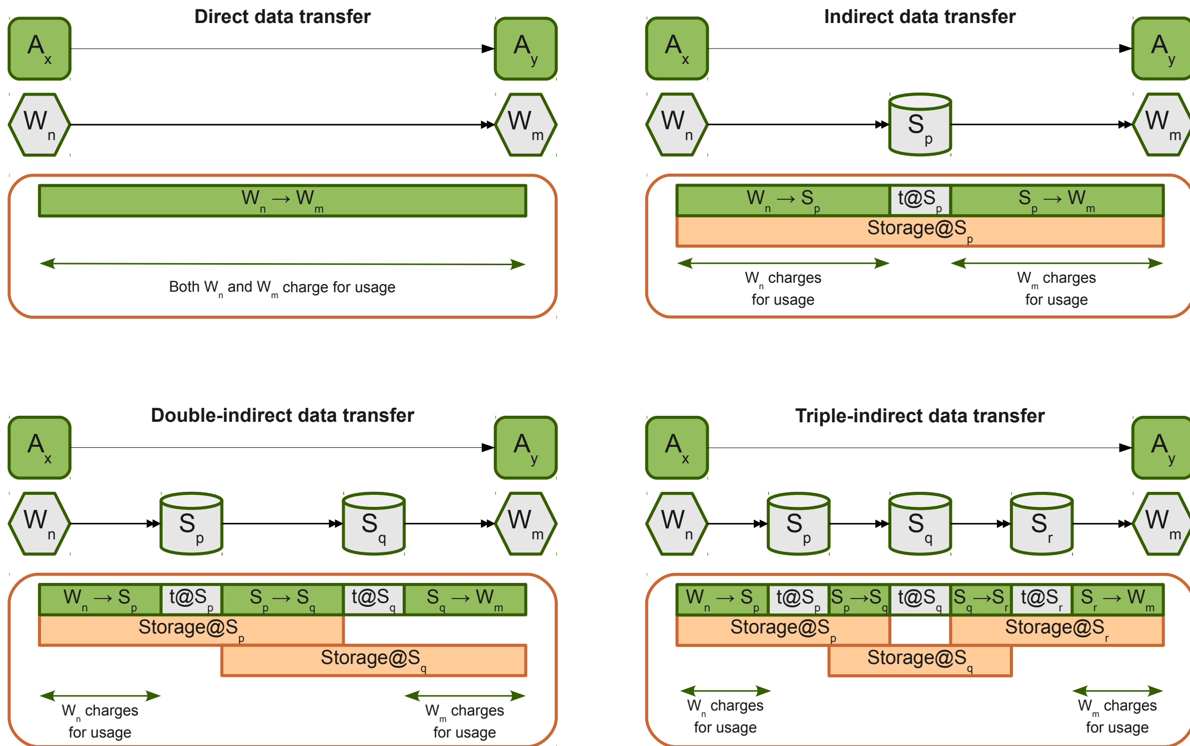


Figure 3.17: Data Transfer Strategy Alternatives

Chromosome Extensions

Which transfer strategy should be employed (and for how long data should be stored at intermediate storage nodes) is usually not obvious, and the optimal choice again depends on user QoS requirements and the state of the infrastructure. However, in principle, we already have all the tools available to simply include this decision in the optimization itself. In other words: if the planning by using a GA is already capable of determining a good schedule, and of partitioning the workflow execution as it does so – why not simply include the determination of the data transfers as well?

In fact, an indirect data transfer, in terms of the resource allocations involved, looks strikingly similar to the allocations involved during an operation invocation. Taking a closer look at Figure 3.17, we see that the resource requirements of all those strategies structurally correspond to co-allocation constraints: in the "top row", we see time-determining allocation constraints, while the storage requirement at the individual SNs

are depicted below, as dependent allocation constraints, each starting and ending with a time-determining allocation constraint (a data transfer, to be exact). The meaning of the resources labeled $t@S_p$ etc. will be further explained below – for the time being, let us simply consider them as some black-box transitory resource.

Adding support for data transfer strategies to the genetic algorithm itself is relatively straightforward: because of the design of the chromosomes, we only need to replace the `DataTransferGene` with a slightly more intricate implementation which allows to choose the strategy itself, and contains genes for the variable parts (i.e., the storage nodes to choose, and the amount of time to intermittently store the data at each node). The approach we implemented is depicted in Figure 3.18, and each of the newly introduced genes is briefly discussed below:

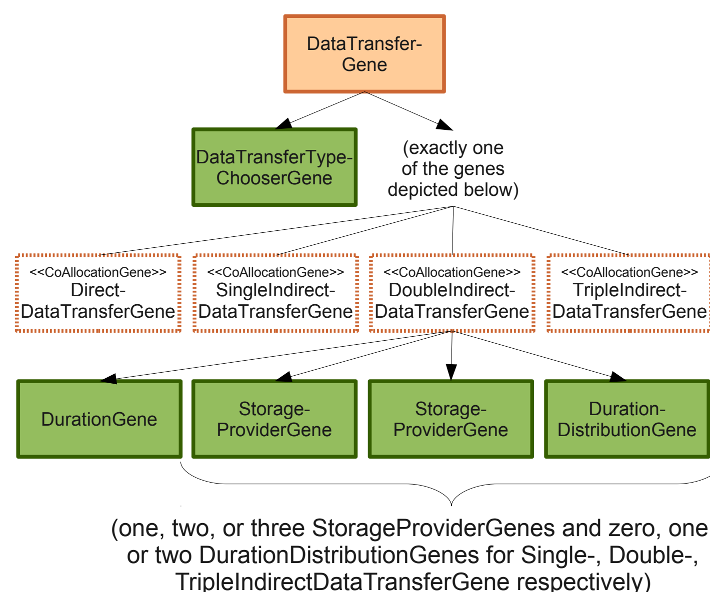


Figure 3.18: Extended Data Transfer Genes Layout

- **DataTransferGene**: this gene is now turned into a composite gene, which contains one gene determining the strategy, and another one representing the actual data transfer.

- **DataTransferTypeChooserGene**: this gene determines the strategy used for the data transfer. The possible allele values are the class names of the actual genes implementing the data transfer. If this value is changed (mutated), the corresponding sibling is replaced by an instance of the chosen class.

- **DirectDataTransferGene**: this gene represents a direct data transfer without using any storage node. In fact, it corresponds to the original implementation of the DataTransferGene, as depicted in Figure 3.3.

- **Single-, Double-, TripleIndirectDataTransferGene**: these genes are composite genes representing the respective indirect data transfer strategy. They share a

common structure[4], differing only in the number of StorageProviderGenes and DurationDistributionGenes.

- **StorageProviderGene**: This gene determines which storage node to use for a particular "hop". Similarly to workflow engines and operation providers, the allele values contain the SOAP endpoint of the respective storage node.
- **DurationGene**: This gene determines the *total* time that storage nodes will be used.
- **DurationDistributionGene**: these genes determine how storage time should be distributed between two consecutive storage nodes. The allele values are real numbers in the interval $[0, 1]$ and are evaluated as described below.

The decision to further decompose the `DataTransferGene` into multiple subgenes may be unusual, but this implementation was chosen on purpose: from a purely practical perspective, it allows to leverage modularity that the hierarchical chromosome representation provides, and allows to reuse the same building blocks to determine simple as well as complex co-allocation constraints. In terms of the outcome, there is no difference between a traditional "monolithic" implementation or the modular one we employed. Internally, evaluating such an extended `DataTransferGene` works exactly as described previously, by using the VDG. The individual `*DataTransferGene`s are actually strategies implementing the co-allocation constraints, which are influenced (i.e., restricted) by the allele values of the subgenes, and whose structure varies as shown in Figure 3.17; ultimately, a compliant co-allocation is returned in all cases.

Storage Duration And Corresponding Resource

The last item which still requires clarification is how the individual durations of using the involved storage nodes is determined, and how they are represented in the co-allocation constraints (and co-allocations). We want the planner to determine a strategy which, in plain words, would be expressed as "Upload the data to the storage node, then leave them there for some timespan *n* (effectively doing nothing), then continue shipping the data". However, the definition of co-allocation constraints requires a sequence of time-determining allocation constraints, each concerning a resource of a transitory resource time.

It is the step between the upload and download which leads to a problem here: effectively, there is no (real) resource which could determine the duration "required" for this step – it is the planner itself which sets (and mutates) this duration. The solution is simple and integrates well with the model: one just defines a new (virtual) resource type $\perp \in \textsc{ResType}$, such that $\text{resClass}(\perp) = \textsc{Transitory}$. Whenever a delay is required, a new (virtual) resource $\perp_x$ of that resource type is employed, where $\perp_x \in \textsc{Resource}, \perp_x := (\perp, 1)$. Assuming that one wishes to wait n seconds, the corresponding time-determining allocation constraint tc is then defined as $tc = (\{\perp_x\}, 1, 1, n)$.

[4]In the implementation, these genes are all subclassing the same class `IndirectDataTransferGene`.

In Figure 3.17, all the gray boxes represent such virtual resources which effectively do "nothing" – except providing the possibility to actually leave the data stored at the storage provider for the corresponding duration.

Let us now quickly relate this with the (internal) chromosome representation: when considering the resulting co-allocations of any indirect data transfer, the sum of all these durations (over all the storage nodes) will correspond to the allele value of the `DurationGene`. Of course, not all durations at all storage nodes need to be equal – their distribution is influenced by the alleles of the `DurationDistributionGene`s. While there is no need to go into further implementation details, we quickly give three examples of how these alleles play together: For a double-indirect transfer with a total wait duration of 500 seconds, a duration distribution of 0.4 would leave the data for 200 seconds at the first storage node, and 300 seconds at the second one. For a triple-indirect transfer waiting a total of 1000 seconds, the distribution values (0.3, 0.7) would result in wait times of 300, 400, and 300 seconds; the same result would also be obtained, for instance, from the distribution values (0. 056, 0.024).

4

Reservation Enforcement

The distinctive feature of the DWARFS system is its awareness of resource requirements, and the necessity to reserve the necessary resources in advance. The previous chapters showed how such requirements are expressed (in terms of co-allocations), and how they can be accounted for during the scheduling of an upcoming workflow execution.

However, one major part of the equation has not been presented yet: of course, it is not sufficient to merely express resource requirements, and to consider them during scheduling. It is equally important to observe the reserved allocations at runtime – in other words, when executing the workflow, the system has to actually behave as predicted at planning time, and control the resources accordingly. While this is important for persistent resources (avoiding to run into resource shortage, for instance a provider running out of disk space), it is even more so for transitory resources: the planning itself is making use of the predicted timing in order to establish the execution schedule. In other words: if a transitory resource such as CPU or bandwidth cannot be provided with the capacity that it was reserved for, the entire execution may fall behind schedule, ultimately resulting in the failure of the workflow execution.

There are multiple entities involved in the execution of a workflow, namely the workflow engines carrying out the orchestration, the operation instances providing the actual functionality of the activities, and possibly storage nodes. While the workflow engines and storage nodes can clearly be categorized as "belonging" to DWARFS itself, this is not really the case for operation instances. In fact, one can argue that a Web Service provider is simply making available some functionality via a standardized interface, and does not even have to be aware of the context of operation calls (i.e., an invocation can originate from a workflow execution, or it could be a standalone invocation). This is certainly true, and would be an argument against the inclusion of DWARFS-specific functionality into "operational" Web Services. On the other hand, resource management is the underpinning of the entire approach presented here, and is a requirement for all involved parties – including operation providers. In other words: since we require extended functionality from operation providers anyway, we may as well provide the tools for them to comply with our requirements.

This chapter discusses how various types of resources can be effectively controlled in order to abide to established reservations. Such enforcement has to take place at all

involved entities (workflow engines, storage nodes, and actual operation providers). While we focus on how the enforcement can be tackled in the environment that we deployed, the approaches presented here are not necessarily limited to such an environment – indeed, the concepts should be portable to other environments without major modifications.

4.1 Environment and Assumptions

The environment that was used for our implementation is based on Java and the Glassfish Application Server. This setup was chosen because of the portability (i.e., independence from underlying Operating System) that Java provides, and because of the widespread use of Java in SOA and Web Services. Glassfish was chosen because it is the reference implementation for the Java API for XML Web Services (JAX-WS), a programming API which in turn is an integral part of Java 6.

Because this chapter discusses how resource enforcement can actually be implemented, there are necessarily considerations and details regarding the particular setup that we used. Again, while the descriptions relate to our specific setup, the concepts will mostly be adaptable to different environments and setups as well.

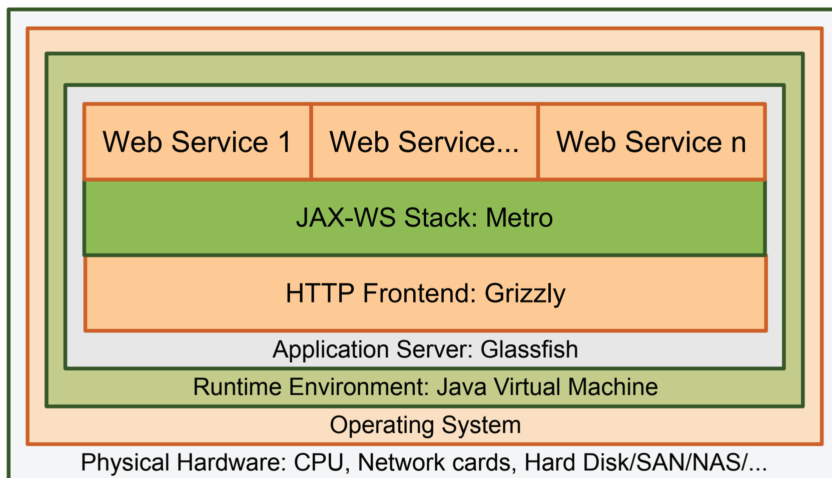


Figure 4.1: Runtime Environment Architectural Overview

Figure 4.1 gives a simplified architectural overview of the runtime environment that we are using. The very first question to answer concerning resource enforcement is: if one needs to control resource usage, at which hardware or software layer does one influence the resource access, and how? There are multiple possibilities, each having their advantages and drawbacks, which we will discuss in the following.

Before going into further details, let us first look at a few other aspects. All of the application logic (independently of whether we are dealing with a workflow engine, an

operation provider, or a storage node), lives inside a web application[1]. What does this mean for the individual resource types? We consider each resource type separately, as there are quite a few differences:

- **Storage:** Storage (i.e., hard disk) access is, in principle, triggered directly by the application logic. Ultimately, all storage access resolves to reading from, or writing to, files (or in the most low-level case, directly from/to a block device).
- **Bandwidth:** From the web application point of view, bandwidth use – i.e., network transmissions – is both happening *outside* of its scope, as well as *inside*. To clarify what this means, consider the case of an operation instance: network transmissions happen before the actual execution (transmission of the invocation request, including its parameters), and after the execution (transmission of the invocation result). The situation is somewhat different for a workflow engine (and storage nodes), as these entities actively start network activity within their own application logic.
- **CPU:** Processing capacity is another case still, because it is essentially used all the time, and even providing the basic capability to execute the application logic in the first place.

Thus, one first aspect to note is that resources may be accessed "explicitly" in the application code (storage/bandwidth), as well as "implicitly" through the framework (bandwidth) or the environment itself (CPU). The second observation is that access and enforcement can be separated. Consider the extreme (and unrealistic) case where the hardware *itself* could determine the reservation pertaining to the context that an access takes place in – in this case, the hardware itself could also regulate its speed (CPU/bandwidth), or keep track of its usage (persistent resources such as storage). While this is not realistically feasible, it shows that enforcement can in principle be performed on a different ("lower") layer of the architecture, as long as the context of the access is known.

We now discuss where enforcement of resource reservation could take place, by referring to Figure 4.1. In principle, each of the different layers (represented by the nested boxes in the figures) could perform the enforcement. We present various properties of the potential solutions for each layer below.

Hardware Layer

Enforcing resource restrictions directly at the individual hardware item would arguably give the most precise control possible. However, it is completely unrealistic to assume such enforcement possibilities, because it would be a prohibitive effort to extend existing hardware with the corresponding features. To mention but one aspect, one would need to define hardware interfaces to convey the restrictions in effect with every device access. Moreover, this feature would have to be supported not only by the hardware itself, but also by the respective operating system drivers, applications etc.

[1]We use the terms "Web Service" and "web application" interchangeably in the context of the implementation inside the application server.

Operating System Layer

An Operating System (OS) is in fact already performing many resource management tasks, including CPU scheduling between the kernel and the running processes, memory (RAM) management etc. It might indeed be the lowest possible layer where reservation enforcement could be done. However, the abundance of different Operating Systems would require an adaptation of each OS (which would probably prove to be impossible for closed-source systems); furthermore, an OS-independent API for representing and manipulating resource reservations would be needed. While theoretically possible, we also deem this a practically infeasible option.

Java Virtual Machine Layer

In this layer, we include both the Java Virtual Machine (JVM) proper, and the standard libraries distributed with the Java runtime, as some of the comprised classes contain native code adapted to or optimized for various OSs. In principle, this is a good place to perform resource enforcement. One could extend the core runtime with classes relating to resources, reservations, enforcement etc. A potential drawback is that one may have to impose the use of a particular JVM onto users, and that one may need to provide a adapted JVM for every supported OS. Moreover, as we shall see in Section 4.5, the controlling of CPU resources would most probably need to be OS-specific.

Application Layer

This layer is the “innermost box” of Figure 4.1, and contains both the Glassfish server (including its sub-components) and the actual application functionality. In other words, the context is a Java process, and interaction is solely happening within the JVM. Therefore, the actual hardware is completely “abstracted away” from the user code, and only accessible indirectly (e.g., via sockets or `java.io.File` objects). On the other hand, if performing the enforcement at this level, one poses the fewest constraints to end-users (and service developers): users can stay with their existing OS and JVM implementation. Ideally, only a few libraries would need to be added to the runtime, not requiring any code changes or other adaptations at all.

We decided to implement the enforcement at this layer, because of the abovementioned reasons. As we shall show in the subsequent sections, a system requiring no changes at all is not practically feasible, but only few guidelines must be adhered to by end users.

4.2 Enforcement of Persistent Resources

A typical example of a persistent resource type, and one which is explicitly defined in DWARFS' formal model, is STORAGE. Resources of this type typically represent secondary storage, e.g., hard disks, SAN, etc.

Controlling the access to persistent resources (i.e., enforcing their usage only within the limits defined in the reservations) is relatively straightforward: one needs to moni-

tor resource access, for example by tracking all write and delete requests, making sure that the total amount used does not exceed the reserved amount at any time. Such enforcement can be done at the application layer (for instance by using a specialized reservation-aware API instead of the standard Java API for performing file access). For the purpose of this thesis and the prototype, we chose to use this simple approach. Another possibility – which we did not investigate further – might be to leverage Operating System support for user-specific quotas. In that case one would need to ensure that all I/O operations take place in the correct (OS user) context.

Other types of persistent resources (e.g., attached hardware instruments) can be handled in a similar way, by enforcing access through methods which verify that the corresponding reservations are in place and obeyed. In the simplest case, it could be sufficient to employ mutexes, semaphores or similar constructs.

4.3 Enforcement of Transitory Resources

As defined in Chapter 2, the class of transitory resource types encompasses all resources which have throughput semantics. Examples of such resource types are BANDWIDTH and CPU. Allocations refering to resources of the TRANSITORY class are compliant with time-determining allocation constraints. As the affected resources have throughput semantics, such allocations are in fact the ones who dictate the duration of co-allocations, and not observing them correctly can lead to a violation of the entire execution schedule. Therefore, enforcing such allocations is more sophisticated than enforcing allocations for persistent resources. In the following two sections, we show how enforcement can be performed for the two transitory resource types which are predefined in the DWARFS model.

4.4 Bandwidth Enforcement

By their very definition, network transfers necessarily have a sending and a receiving end. In other words, an allocation for a bandwidth resource is never a “standalone” allocation – rather, there are multiple parallel allocations with identical usage blocks, but referring to resources at different providers. We denote the resource provider acting as the sending end as *sender*, the one at the receiving end as *receiver*, and all other (possibly) involved resource providers as *intermediates*.

Because these allocations have all been successfully reserved, they can necessarily be accomodated by all involved resource providers. Stated differently, even if the individual resources have different capacities, the allocations are such that their usage never exceeds the capacity of the lowest-capacity resource.

Before we further describe *how* the enforcement is done, let us examine *where* it is sensible to apply the enforcement. In fact, there are only two theoretical cases which need to be taken into account:

- **Enforcement at sender:** the sender regulates the rate at which data is transmitted, ideally at all times conforming exactly to the transmission rate (i.e., usage

amount) specified in the allocation. Transmitting data too fast may result in network congestion. Conversely, transmitting data too slow would result in falling behind schedule with the transfer – which would either make the transfer take longer than allowed by the allocation, or, to make up for the delay, would require to raise the transmission rate beyond the allowed value.

- **Enforcement at receiver or intermediate:** any resource provider which receives data can only receive them after they have been sent. At this point in time however, enforcement attempts are no longer useful: If data had been sent too slowly, there is nothing the receiver can do about it. Similarly, if data had been sent too quickly, then it either arrives before schedule (resource capacity permitting for this), or it may already have caused a network congestion. Thus in any case, attempts to regulate resource usage at a receiver or intermediate are practically futile.

Effectively, enforcing bandwidth allocations resolves to throttling the rate of outgoing data at the sender's side. Implementation-wise, this means that one needs to control the rate of writing to the OutputStream object representing the socket connection. We then employ a simple method: since the complete allocation is known, we can determine, for any timestamp, the total number of bytes that should have been sent at that time, and we know how many bytes have actually been sent. We then throttle throughput accordingly.

4.5 CPU Enforcement

The second transitory resource type that the DWARFS model supports is CPU, representing the "computational power" of an operation instance. In Chapter 2, we have stated that it is the resource provider's duty to specify the exact capacity and units for resources of this type. The example given in that section mentioned using the "operations per second" metric as an example, as many CPU manufacturers provide such information in their data sheets. For our prototype implementation, we have simply measured how many iterations of a CPU-bound activity the machine can execute within 5 minutes, and then scaled that value down accordingly.

Independent of the actual capacity, one can also express CPU usage using percental values of the capacity. This is a widespread representation to visualize system load in most operating systems, as also depicted in Figure 2.3 (p. 10). For reasons further explained below, using such relative values is also the most useful approach for implementing CPU enforcement.

The approach to monitor and enforce CPU reservations is as follows: at the operation provider, each operation invocation is processed by running one or more Java threads. Such a "bundle" of threads is called a *task*. Because in DWARFS, all required resources must be reserved in advance, each task is thus necessarily executed within the context of a reservation encompassing allocations for CPU resources. A *supervisor* component is aware of all running tasks and their associated percental CPU allocations (called *share*

for short). The supervisor constantly monitors CPU usage of these tasks, and regulates them accordingly to enforce the CPU usage to adhere to the allocations.

From a Java program, interactions with the system scheduling are rather limited: to retrieve information about CPU usage, one can only query the total amount of CPU time (in nanoseconds) that each considered thread has used. Similarly, to influence the scheduler, one can only set thread priorities to one of the 10 Java priorities (or, in extreme cases, suspend and resume threads). With only these two instruments at hand, is it possible to force individual tasks to their requested share?

The above question was in fact one of the earliest topics tackled during the development of DWARFS, because it is a *condicio sine qua non* for the entire approach: if it is not possible to influence the priorities of concurrently executing operations on the provider side so that reservations are met with sufficient accuracy, then advance reservations become pointless as well.

4.5.1 Gathering CPU statistics

As stated, the only way to gather CPU usage information from within Java consists of retrieving, for each thread to consider, the total amount of CPU time spent in that thread. For multi-threaded tasks, one simply sums up all values to determine the total share. But moreover, there is also no way to determine (from within a Java program) the theoretical CPU capacity. In other words, one can only rely on the actually measured data. Thus, to determine the effective CPU percentage of a thread, one must first sum up all considered threads' used CPU times to determine the 100% ratio, and only then can one calculate the actual shares.

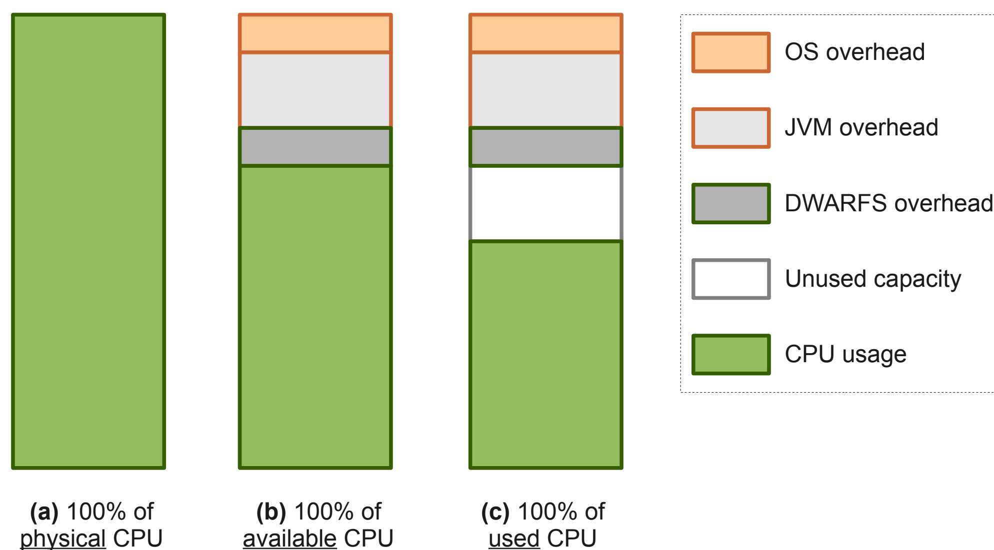


Figure 4.2: CPU shares and overhead

Figure 4.2 depicts this relationship, as well as the overhead introduced by various other parts of the system. Figure 4.2 (a) represents 100% of the physical CPU available, and serves as a reference. In Figure 4.2 (b), the overhead introduced by the Operating

System, the Java Virtual Machine, and framework itself are depicted. Intuitively, the bottommost bar represents what is available to the "operational" threads (the ones actually representing supervised tasks), and what would be seen by the supervisor if the system was fully loaded. Finally, Figure 4.2 (c) shows what the supervisor would see as 100% if the system was not fully loaded, for instance if a supervised task was running in a single thread on a multi-core machine. This is caused by the way the calculations are performed, as explained above. However, because we only rely on relative shares, the reasoning is still correct regardless of the actual load on the system. Note that the figure is not drawn to scale, but purely illustrational – while we cannot reliably measure the OS and JVM overhead, we expect them to be rather low, and our measurements have shown that the overhead of the supervisor, in terms of CPU usage, is negligible.

4.5.2 Influencing CPU shares

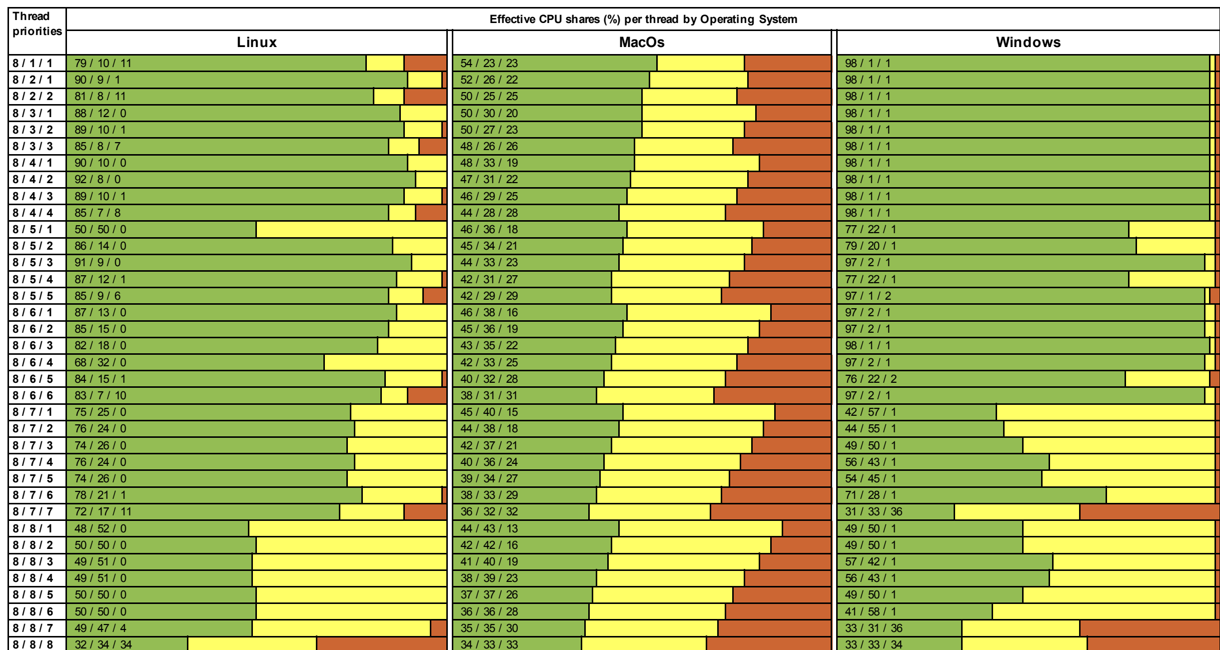


Figure 4.3: Mapping of Java thread priorities to effective CPU shares on different OS's

As in Java, the only method to influence the CPU resources that a particular task gets is to modify its thread priorities, our first experiments consisted of observations of how multiple concurrent threads running with different thread priorities behaved in terms of the assigned CPU shares. Figure 4.3 shows a representative part of these experiments, where three threads were run in parallel, with the priority of the first thread fixed to 8, and the other two threads taking all possible combinations of priority values from 1 to 8. The resulting CPU shares are depicted textually and graphically, where each thread is represented by a different color.

The experiments were run on the same laptop using a triple-boot setup with Mac OS X 10.4 (64 bit), Ubuntu Linux 8.04 (64 bit), and Windows XP (32 bit). In all cases, a Sun

JVM 1.6 was used, and the systems were running natively and not otherwise loaded. The results are quite surprising and clearly show fundamental differences between the underlying OS schedulers. Put bluntly, only Mac OS seems to produce somewhat sensible results (where the proportions between the effective shares are somewhat reflecting the thread priorities), while both Linux and Windows were disproportionally penalizing low-priority threads.

Another, more fundamental, issue arises from the coarse granularity of the thread priorities themselves: Even assuming a perfectly proportional mapping of priorities to CPU shares, a share distribution of 5%/95% between two tasks would not be representable using any priority combination.

The fact that not all possible requested share combinations can be accomodated by a fixed combination of Java priorities (thus requiring adjustments at runtime), and the rather big differences in behavior of the schedulers among different OS's were the main motivation for building a controller which continuously evaluates the current state, and dynamically adjusts the task priorities at runtime.

4.5.3 The Control Loop

The problem that we must tackle here is indeed a classical case encountered in control theory. One possible approach to control the system is to use a variant of a PID controller [AH95]. Very briefly, a PID controller knows the current and the target system state, and considers the current error (termed proportional – P), the sum of the past errors (termed integral – I), and the rate of change of the past errors (termed derivative – D). Based on a function of these error values, the system state is influenced in an effort to drive it to the target state, i.e., to minimize the current error. We had in fact started with an implementation of a PID controller initially, but because it was rather difficult and unintuitive to "fine-tune" the parameters to get the desired accuracy, we soon abandoned it for the solution described below.

When taking one step back from the implementation and looking only at the concept of the controller, one can very easily express – in natural language – the actions that it should take. For example, if one was to manually control the thread priorities, one could act along a rule set such as: i) if a task is getting too few CPU resources, and has been doing so repeatedly, then drastically raise its priority; ii) if a task is constantly getting exactly the share it requested, then do nothing; etc. The actual implementation, which uses a fuzzy controller, is very closely related to such intuitive descriptions. As the following explanations are illustrated with concrete examples, it is necessary to quickly introduce the employed terms:

- **badness**: refers to how well a particular task abides to its requested CPU share. The system may have overspent CPU for a given task (resulting in the task getting more compute power than reserved, and thus a positive badness values), or it may have underspent (resulting in negative badness).

- **tendency**: corresponds to the trend of a task's badness; i.e., it captures whether the badness value is rising or dropping. It is in fact a function of the derivative of the badness.

- **action**: represents the output of the controller. This value influences how a task's priority is adjusted (lowered or raised).

The subsequent sections will gradually enhance these rather short descriptions, to give the complete picture.

4.5.4 Fuzzy Logic and Fuzzy Controllers in a Nutshell

This section gives a very condensed and rather informal overview about the concepts behind fuzzy logic and fuzzy controllers. Much more comprehensive material on the topic can for instance be found in [Ros04]. In principle, fuzzy logic differs from "classical" boolean logic in one crucial, aspect, namely that it is many-valued. Fuzzy reasoning is approximate rather than exact, and truth values are not considered to be binary (`false` / `true`, or `0` / `1`), but can take any continuous value between 0 and 1.

Consider the question whether a certain temperature is perceived as hot or cold. While this is of course a highly subjective decision, one would expect the general consensus to be that a temperature of 0 °C is perceived as cold, while 80 °C are perceived as hot. But what about 25 °C ? One could argue that this qualifies as neither hot nor cold, or alternatively as "still a little bit cold, but already somewhat hot".

Crisp Variables

In fuzzy logic terminology, a crisp variable has exactly one, well-defined value – in the example above, the temperature of 25 °C is a crisp variable.

Fuzzy Terms and Sets

Fuzzy logic accounts for such imprecisions that we encounter in natural language. The abovementioned concepts such as "hot" and "cold", in fuzzy logic, are called fuzzy terms or linguistic variables. Generally speaking, a fuzzy term is described by a function whose domain depends on the concept itself (in this case degrees Celsius), and whose codomain is the interval $[0, 1]$, representing the truth value, or confidence. Seen from another perspective, a fuzzy term describes a semantic concept and identifies which values belong to that concept by "how much". Fuzzy terms are merely special cases of the general notion of *fuzzy set*, whereby a fuzzy set F is mathematically described by a membership function μ_F, mapping from some domain to $[0, 1]$.

Figure 4.4(a) shows an example of how temperature-related linguistic variables – in this case, the terms are *cold*, *warm*, and *hot* – may actually be defined. It is important to note that fuzzy terms can, and usually do, overlap: in the example, a temperature of 12 °C would qualify as *cold* with a truth value of 80%, and at the same time as *warm* with a truth value of 15%.

The fuzzy terms for the previously defined variables (badness, tendency, and action) are presented in Figure 4.4(b-d). The names and the definitions of the membership functions should illustrate how helpful many-valued logic can be to express subjective and imprecise or vague facts.

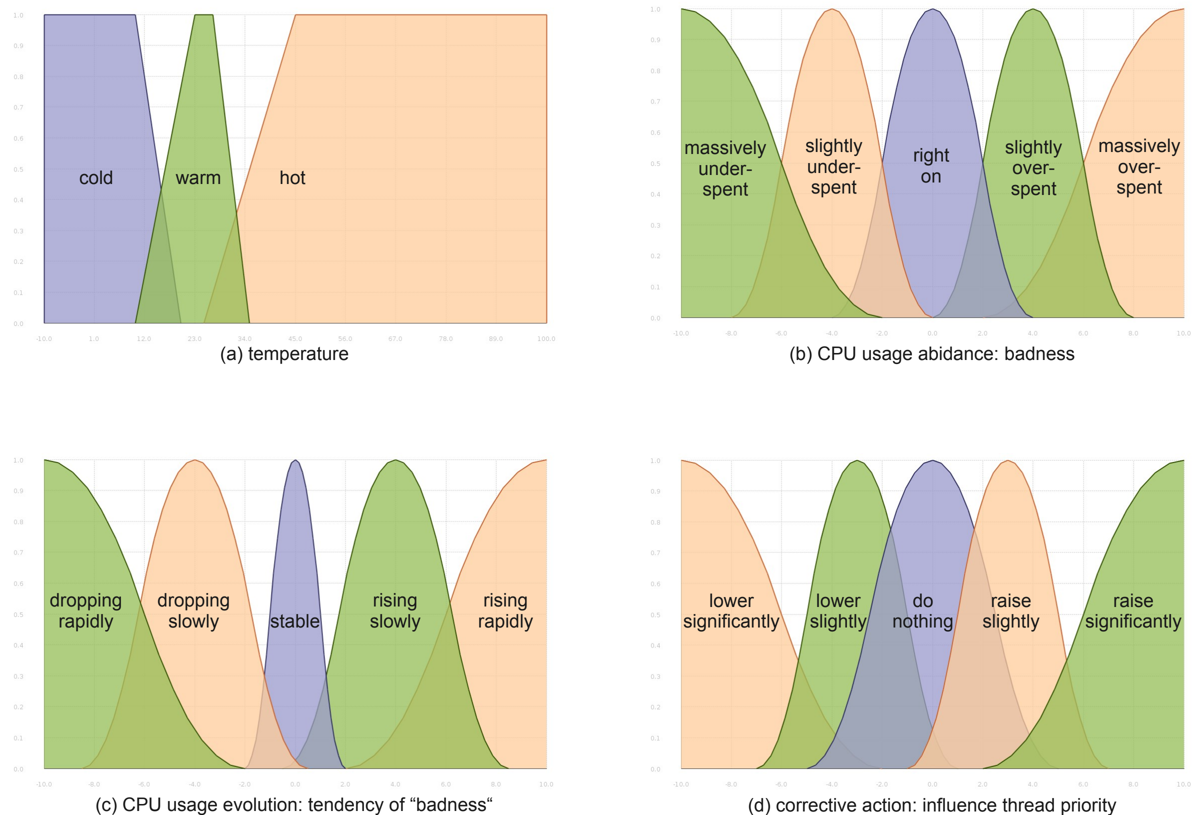


Figure 4.4: Fuzzy Term Definitions

Fuzzification

In simple terms, fuzzification refers to the transformation of a crisp value into grades of membership of linguistic variables. In fact, we have already encountered fuzzifications in the text above, namely:

$$\mu_{cold}(12°C) = 0.8$$
$$\mu_{warm}(12°C) = 0.15$$
$$\mu_{hot}(12°C) = 0$$

Fuzzy Logic Operations

Logical operations present in boolean algebra have their counterparts in fuzzy logic as well. Common definitions of the most basic operations are given below.

- conjunction: a `and` $b := \min(a, b)$.
- disjunction: a `or` $b := \max(a, b)$.
- negation: `not` $a := 1 - a$.

In fact, these are the exact same defitions as in boolean logic, except that the truth values are continuous in the fuzzy case.

Fuzzy Set Operations

The above operations can be "extended" to entire fuzzy sets as well, by applying them to the complete fuzzy set domain. For instance, the temperatures that are considered "*warm* AND *hot*" are the ones contained in the intersection of both sets – which can again be expressed in terms of a function, as shown below:

- conjunction: A AND B ($A \cap B$): $\mu_{A \cap B} := \min(\mu_A, \mu_B)$.
- disjunction: A OR B, or $A \cup B$: $\mu_{A \cup B} := \max(\mu_A, \mu_B)$.
- negation: NOT A, or $\bar{A}$: $\mu_{\bar{A}} := 1 - \mu_A)$.

Note that these set operations are only meaningful when applied to fuzzy sets whose defining membership functions have the same domain. As an example, it is not reasonable to (try to) determine the fuzzy set $overspent_low \cup warm$.

Fuzzy Rules

By combining and using the above prerequisites, one can define a number of *fuzzy rules* such as:

```
IF   badness IS overspent_low AND tendency IS rising_rapidly
THEN action = lower_lot
```

The first line, i.e., the condition, is termed *antecedent*, while the conclusion is termed *consequent*. The Figures 4.5 and 4.5 present how the implemented rules would handle two particular situations. They depict the entire flow of the rule evaluation from crisp input values to a crisp output value.

Antecedent evaluation (depicted on the left side of each figure) consists of the fuzzification of the crisp values obtained as input from the system, and of applying fuzzy logic rules to the resulting truth values. The figures show only the relevant subset of all implemented rules, i.e., only the antecedents which yield non-zero truth values are depicted in both cases.

During fuzzy reasoning, all rules are evaluated in parallel, and the final result is obtained from the union of all consequents (depicted on the right), as described below.

Determination of Consequents

Each antecedent produces a single truth value, representing the overall fulfilment of the rule's condition. As the consequent is defined by a linguistic variable, it is natural to apply this fulfilment to the consequent as well, thus "capping" it at that value by means of a simple AND (min) operation.

Just like all antecedents are evaluated, all consequent results are finally aggregated. Since all of the consequents contribute to the final result, the aggregation is performed as a set union, i.e., using an OR (max) operation.

Figure 4.5: Fuzzy rules evaluation: overspent CPU with dropping tendency

Figure 4.6: Fuzzy rules evaluation: overspent CPU with rising tendency

Defuzzification

The final step of the rule processing is to produce a crisp output value from the obtained fuzzy set. This is generally done using centroid defuzzification, whereby the (value of the) set's center of gravity is returned. This corresponds to the idea that this value captures the best "compromise" of all individual rule evaluations.

Summary

This section was meant to both give a very brief summary of fuzzy logic and fuzzy controllers, as well as to present the most important aspects of our system. The results which are given subsequently were achieved using this configuration.

The controller we use is actually a generic fuzzy controller built for the purpose of, but not limited to usage in, DWARFS. It is completely (re-)configurable at runtime (i.e., all the logic performed, such as getting or setting the system state, fuzzification of parts of it into fuzzy values, fuzzy rule evaluation, and defuzzification, is configured declaratively), and supports detailed logging of the system state. A User Interface provides users with the ability to perform the configuration, as well as to "replay" and single-step through logs for analyzing them.

4.5.5 Implementation Details

Maximum Task Share Calculation

Each operation invocation will result in one or more threads running, which we define as constituting a *task*. With multi-core machines, an additional factor has to be taken into account: On a machine with P cores, the maximum achievable share of a task t with n threads is $s^t_{max} = \min(1, \frac{n}{P})$. If the system allowed reservations for more than s^t_{max}, the task could not be able to achieve the expected share, resulting in an erroneous slowdown of other simultaneously running tasks. For instance, on a dual-core CPU with two threads running at full speed, each thread will run on one core – a reservation combination of 70%/30% will result in the first thread never being able to achieve its envisaged goal, but to be blocked at (a maximum of) 50%. The second thread, however, is also not abiding to its 30%, because no matter how low the priority is, the thread will utilize the otherwise unused CPU and run at 50%. It is therefore crucial to know s^t_{max} for a given operation before accepting a reservation request for s^t_{req}, so that these limits can be enforced. This results in the requirement to know the number of threads a given operation will run.

Monitoring and Control of CPU Shares

Whenever a new task (i.e., operation call) is started, the system determines the necessary metadata. It then periodically performs the following calculations to monitor and control execution for the set T of currently active tasks:

- Calculating the current expected shares (s^t_{cur}) of all tasks, and adjusting them so that $\forall t \in T : s^t_{req} \leqslant s^t_{cur} \leqslant s^t_{max} \wedge \sum_{t \in T} s^t_{cur} = 1$. Note that this implies that tasks may get more resources – and thus finish faster – than requested. The objective is to avoid tasks getting too few resources.

- Gathering the CPU usage, and computing the actual share $s^t_{act} \forall t \in T$. So, while s^t_{cur} represents the currently expected share for a task, s^t_{act} is the currently measured share.

- Passing s^t_{act} and s^t_{cur} to the fuzzy controller, and possibly adjusting the thread priority for all of the task's threads in response to the controller output.

Note that the system does not address a "full-fledged" scheduling problem (i.e., it neither has to, nor wants to, assign exactly which tasks have to be run at which moment, which anyway would require to entirely replace the OS or the JVM scheduler). Instead, it merely modifies the priorities of tasks so that their overall CPU consumption matches the requested one.

Predicting execution times

After a task has finished, the logged information about elapsed times and CPU usage can in turn be used for future predictions of the operation: If task t had run for n intervals with different expected shares (s^t_{cur}), its overall execution E_t can be represented as a set of n time slices $\tau_i = \langle \delta_{\tau_i}, \sigma_{\tau_i} \rangle$, where $\delta_{\tau_i} \in \mathbb{N}^+$ is the duration of the ith slice, and $\sigma_{\tau_i} \in (0, 1]$ is the corresponding actual CPU usage. The predicted execution time for t is then calculated as $PE_t = \frac{1}{s^t_{max}} \sum_{i=1}^{n} \delta_{\tau_i} \sigma_{\tau_i}$. This prediction can be linearly scaled if shares other than s^t_{max} are requested.

For evaluation and comparison purposes, we repeatedly (15 times) ran the following configuration: The same CPU-intensive operation (repeatedly calculating SHA-512 hashes, as a representative of a purely CPU-bound and expensive calculation) is run as 6 different tasks, started at different times and with varying requested priorities. This setting was chosen since it contains most of the interesting aspects of a real-life setting, i.e., tasks starting at "random" times (also at the same time), high-priority tasks intercepting lower-priority ones, tasks acquiring additional (otherwise idle) CPU resources, etc. All tests were performed on the same computer, running on Ubuntu 8.04 (64-bit) and Windows XP SP2 (32-bit), in a normal, not otherwise loaded configuration. In all cases, a Sun JVM 1.6 has been used, and the control loops were effectuated every 500 ms.

Figure 4.7 depicts the evolution of the system state, as seen from the controller. For each task t, s^t_{cur} (*should*) and s^t_{act} (*is*) are depicted. An important point is that *should*-values are adjusted as tasks join and leave, defining the slice boundaries and resulting in a stair-case-like *should* curve. The controller tries to keep *is* as close to *should* as possible. The oscillations at the boundary start are caused by the fact that each adjustment of the target values (*should*, or s^t_{cur}) results in the need to take the boundary as the new starting

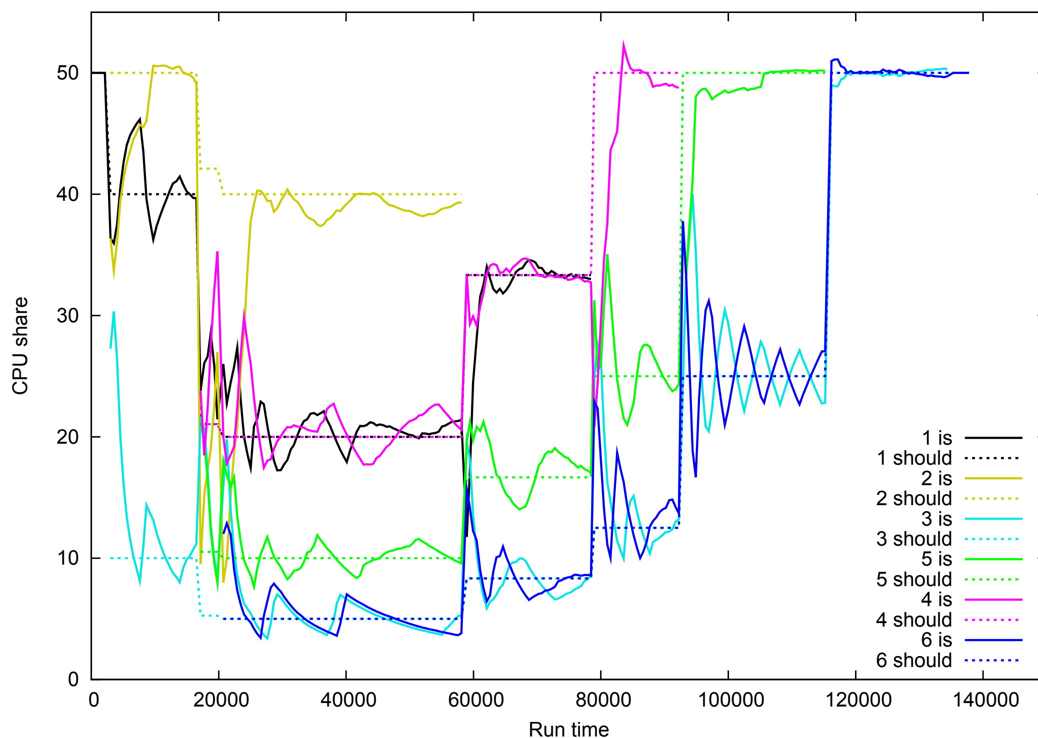


Figure 4.7: System state evolution during CPU share controller run

point for share calculation, thus starting the calculations “from scratch”. Naturally the resulting coarse granularity of input data, paired with few reference intervals, cause a greater imprecision in the calculations and therefore peaks in the representation. In fact, a more intuitive representation of the system state – and more insight into the effectiveness of the controller – is gained by accounting for the performance during previous timeslices, which is done by calculating *as* and *ai* as the average of all *should* (respectively *is*) values over the complete lifetime of the task. These aggregated values are depicted in Figure 4.8.

Table 4.1 presents the evaluation of our measurements. The uncontrolled execution time corresponds to the task being run as a standalone application outside of the controller and serves as a control variable. While we cannot explain the striking difference in execution times between Windows and Linux (possibly caused by the difference between 32 and 64 bit mode), it is actually helpful for analyzing the effect of longer task run times.

The most important functional quality criteria are the absolute and relative errors, which correspond to an inability of the system to enforce the requested reservations. The results show that it is indeed possible to enforce reservations, with the quality of the enforcement and the predictions improving with the duration of a task. The price to pay is a performance penalty, as shown by the predictions. The predictions are generally slower than in the uncontrolled case, as (mostly low-priority) tasks overspending their assigned shares have to repeatedly be suspended so that other tasks meet their target shares.

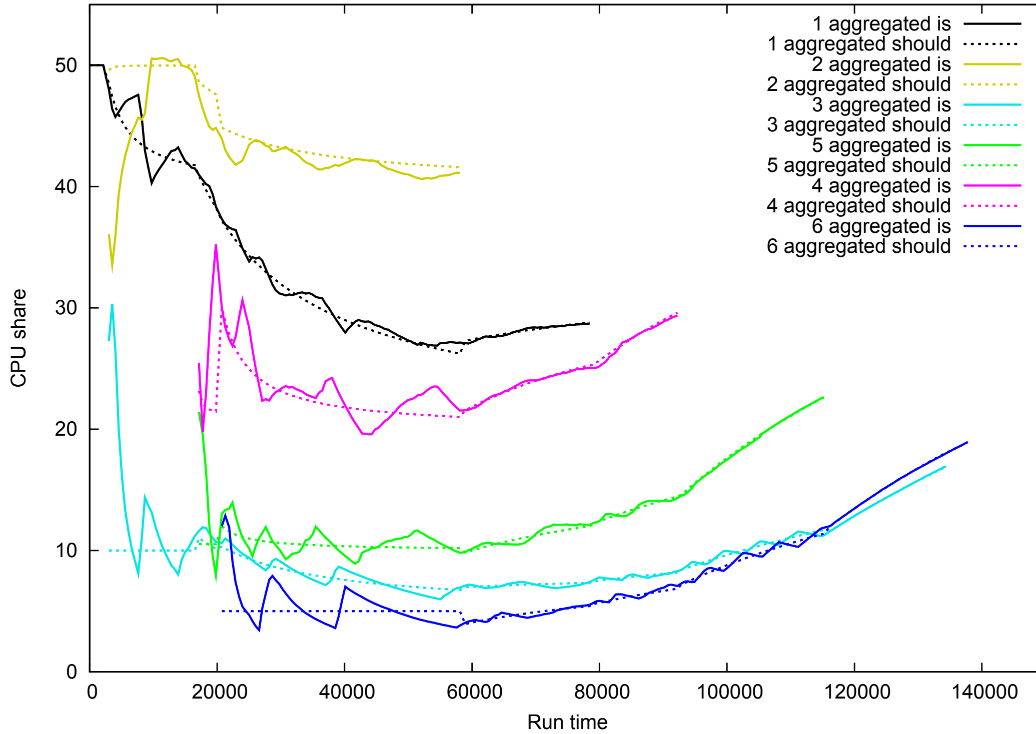


Figure 4.8: System state evolution (aggregated shares)

Table 4.1: Evaluation results

Item (averaged over 15 runs)	*Windows*	*Linux*
Uncontrolled execution time (ms)	216829	41361
Coefficient of variation for uncontrolled ex. (%)	0.99	2.19
Predicted execution time (ms)	221290	48502
Coefficient of variation for prediction (%)	2.05	5.01
Factor prediction/uncontrolled	1.02	1.17
Average absolute deviation $\|ai - as\|$ (%)	0.63	0.79
Average relative error $\frac{\|ai-as\|}{as}$ (%)	5.41	5.96

Please note that the abovementioned short evaluation only focused on the enforcement of CPU shares for concurrently running tasks. A more comprehensive evaluation of the enforcement at the scale of entire workflows in a complete Service-Oriented Infrastructure will be given in Chapter 6.

5

Infrastructure Implementation

While the previous chapters on Planning and Execution already described some specific implementation aspects, this chapter gives the "large scale" overview of our prototypical implementation of the entire DWARFS infrastructure.

5.1 Infrastructure

As mentioned previously, we strived to leverage existing and widely adopted technologies wherever possible and sensible, extending them with DWARFS-specific functionality where required. Therefore, we decided to use the widespread WSDL/SOAP Web Services standards as a basis for the implementation of the functionality. As the implementation is based on Java, which – starting with version 5 – includes Web Service support directly in the core class library, we used Jax-WS as the Web Services API, both client- and server side.

While multiple application servers are available, we chose the Glassfish application server, because a) it is widespread and mature, b) it is the de-facto reference implementation for Jax-WS-based application servers, and c) it supports much of the required functionality (such as MTOM for streaming large input and output data).

Finally, information about available services is kept in one or more UDDI registries.

Figure 5.1 presents a high-level overview of what a DWARFS infrastructure might look like. The rest of this section will present in more detail the different components which make up the infrastructure, and their interactions.

5.1.1 Terminology

This chapter is looking at the DWARFS system from a systems implementation perspective, and considers existing technologies which have their own established terminology. While much of that terminology can be directly mapped to the terms used in the System Model (Chapter 2), that model focuses on the crucial aspects, and does not contain definitions for some of the concepts which are present in this chapter. Conversely, the

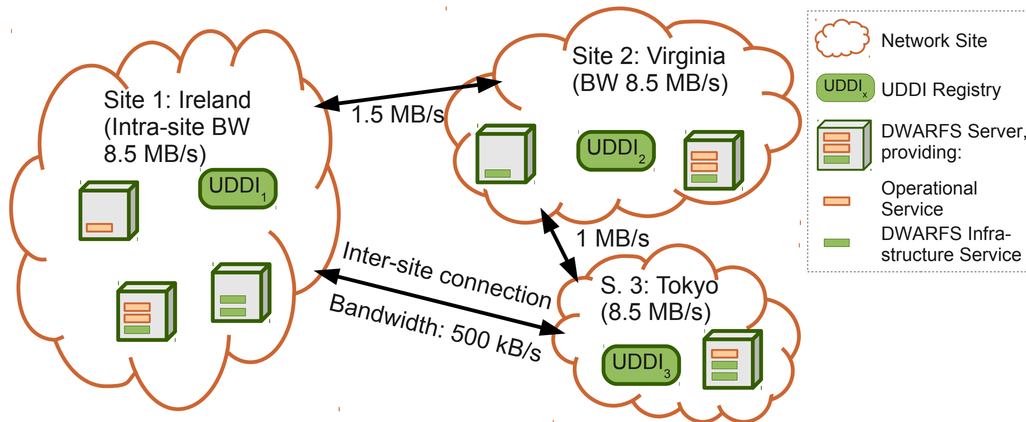


Figure 5.1: Infrastructure Overview

model also defines some terms which are represented differently at the infrastructure layer. We will describe such cases in this section.

Operations, Operation Instances, and Endpoints

In the System Model presented in Chapter 2, we defined the terms OP (Operation) and OPINST (Operation Instance), where – informally speaking – the former is the definition of some functionality, while the latter is an implementation of that functionality, deployed on an ENDPOINT. The SOAP/WSDL specifications introduce a few more entities, namely:

- **Port Types**: A port type is simply a set of operations. In Object-Oriented (OO) terminology, this would closely match the concept of an Interface.
- **Bindings**: A binding associates a port type to a particular transport protocol. In other words, it prescribes the method to serialize data when it is physically sent.
- **Ports**: A port provides information about where a particular binding can be physically invoked. In the case of DWARFS, the deployment information is represented as an HTTP URL.
- **Services**: A service is the aggregation of one or more ports. In OO terminology, this would correspond to an object instance implementing at least one interface.
- **Application Servers**: An application server is an instance of – in our case – a Glassfish server. It provides one or more services. In the following, we will use the term **DWARFS Server** to designate an application server which provides the basic DWARFS components (see below).

To summarize the correspondence to the Model terminology: a DWARFS Server can be understood as representing an ENDPOINT, while (possibly multiple) Operations are defined in a binding, and (possibly multiple) Operation Instances are deployed as a service. We designate such services as **Operational Services**.

DWARFS Infrastructure Services and Components

We have just discussed the organization and deployment of Operational Services (i.e., the services providing the "target" operations for a workflow). However, there is a lot of other, system-specific functionality to be implemented – most importantly, DWARFS requires Workflow Engines to actually orchestrate executions, and it requires functionality to handle resources and reservations, data and invocation characteristics, etc.

In order to stay consistent with the overall architecture, such functionality is also implemented as Web Services (or as libraries which are to be used from within Web Services). More details are given below.

5.1.2 Components Overview

This section describes the components depicted in Figure 5.1. In particular, we will also go into more detail about the services that can (or must) be deployed on individual DWARFS Servers.

Network Site

Network sites are not actually physical components, but rather organizational and logical entities. They simply represent the physically separated (and individually administered by their owning organizations), but interconnected networks which exist in the real world. There are two assumptions which will generally (though not necessarily always) hold:

- An organization (or organizational unit) has full control over the hosts and networks present in one site, but normally does not have administrative privileges for other sites.
- Servers co-located at the same site usually have better intra-site connectivity – in terms of network speed – than with other sites.

UDDI Registry

A UDDI Registry contains information about the services which are present in the infrastructure. As stated above, in a real-world setting where multiple organizations jointly provide the infrastructure, there are administrative boundaries typically limited to the individual network sites. This fact is depicted by the presence of one registry at each site in 5.1. More specific information about what information is present in a UDDI registry, and how it is represented and queried, is provided in Section 5.3.1 below.

While multiple registries are supported, our prototype implementation uses a single one. We use an Apache jUDDI ([Thea]) instance deployed on a Tomcat [Theb] application server.[1]

[1] The necessity to use an additional application server software arose because jUDDI cannot be easily deployed inside a Glassfish container.

DWARFS Servers

There may be an arbitrary number of servers in a DWARFS infrastructure, each providing an arbitrary number of Web Services, as shown in Figure 5.2. While we chose Glassfish as the Application Server, in principle, any compatible server software could be used.

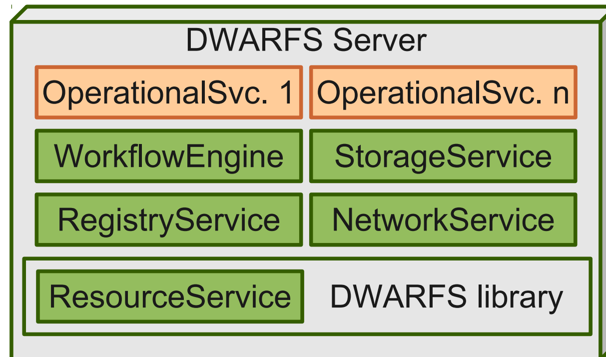


Figure 5.2: Components deployed on DWARFS Servers

All subsequently discussed components are components which can be deployed on DWARFS servers.

DWARFS Library

This is a mandatory library which must be installed on every DWARFS Server. It provides common functionality required by DWARFS services running on the server, such as Resource and Reservation management, configuration retrieval, registration of deployed services with a Registry Service, and enforcement of reservations.

Resource Service

The Resource Service exposes a Web Service providing resource management functionality. In particular, it is the (only) service which provides operations to query and modify resource reservations. It is thus mandatory to deploy this service on every DWARFS server.

Registry Service

This service provides a simplified interface to a UDDI Registry. It allows other DWARFS services to register themselves with the registry. There must be at least one Registry Service in the infrastructure – though in a real-world setting, one would expect one at each network site, configured to interact with a particular UDDI registry. Individual DWARFS services must be configured with the Registry Service to use (or rather: the URL for the Registry Service's WSDL document).

Network Service

While each DWARFS Server autonomously performs the Resource and Reservation management for the resources associated with it (i.e., CPU, Storage, incoming and outgoing Bandwidth), both intra- and inter-site network connections do not have a "natural" endpoint which can be associated to them. It is therefore necessary to explicitly designate a particular Network Service, which manages the corresponding resources. A network service can manage the resources for an arbitrary amount of networks. There may be zero or more Network Services deployed in a DWARFS infrastructure. If no Network Service is deployed, then network transfers are not considered as "manageable" resources. If multiple Network Services are deployed, then they must manage disjoint sets of resources.

Workflow Engine

If processes are to be planned, or executed, in a DWARFS infrastructure, at least one Workflow Engine must be deployed. To leverage the full potential of distributed workflow execution, especially when multiple sites are involved, it is advisable to deploy one or more Workflow Engines at each site.

Storage Service

A Storage Service can temporarily store data. Deploying such services is not mandatory, but if they are present, they may allow to optimize data transfers as described in Section 3.5.

Operational Services

The DWARFS Server components and services mentioned above are all related to the DWARFS infrastructure itself – or, more precisely, they *constitute* the infrastructure. The services which actually provide the domain-specific operations that workflows consist of are referred to as Operational Services. Their actual implementation is generally out of scope for the DWARFS infrastructure. However, these services will also require some of the functionality provided by the DWARFS library – they must register themselves with a Registry Service on startup, they must provide the required metadata, such as invocation characteristics, and they must use the enforcement functionality provided by the DWARFS library during operation invocation. Section 5.3.3 below contains further details about the practical aspects of providing an Operational Service for the current implementation.

5.2 Interactions

While the previous section presented the components constituting the DWARFS infrastructure, this section focuses on how these components interact. While the figures in this section use a sequence-diagram-like representation to ease the understanding, they

are not to be (strictly) interpreted as sequence diagrams. Instead, they focus on the most important interactions, but omit some of the less important details.

There are three different use cases which are particularly interesting in terms of interactions, namely Service startup and registration, Workflow Planning, and Workflow execution. We will describe each of these scenarios below.

5.2.1 Service Startup and Registration

Because the UDDI registries are the central point of information about the services available in a DWARFS infrastructure, they must be kept up-to-date. While manual maintenance is theoretically possible, it is generally preferrable to have the services themselves "announce" their availability. Figure 5.3 shows the actions which take place when a service starts up. Note that these apply to all services deployed on a DWARFS server, i.e., both Operational Services and DWARFS Infrastructure Services.

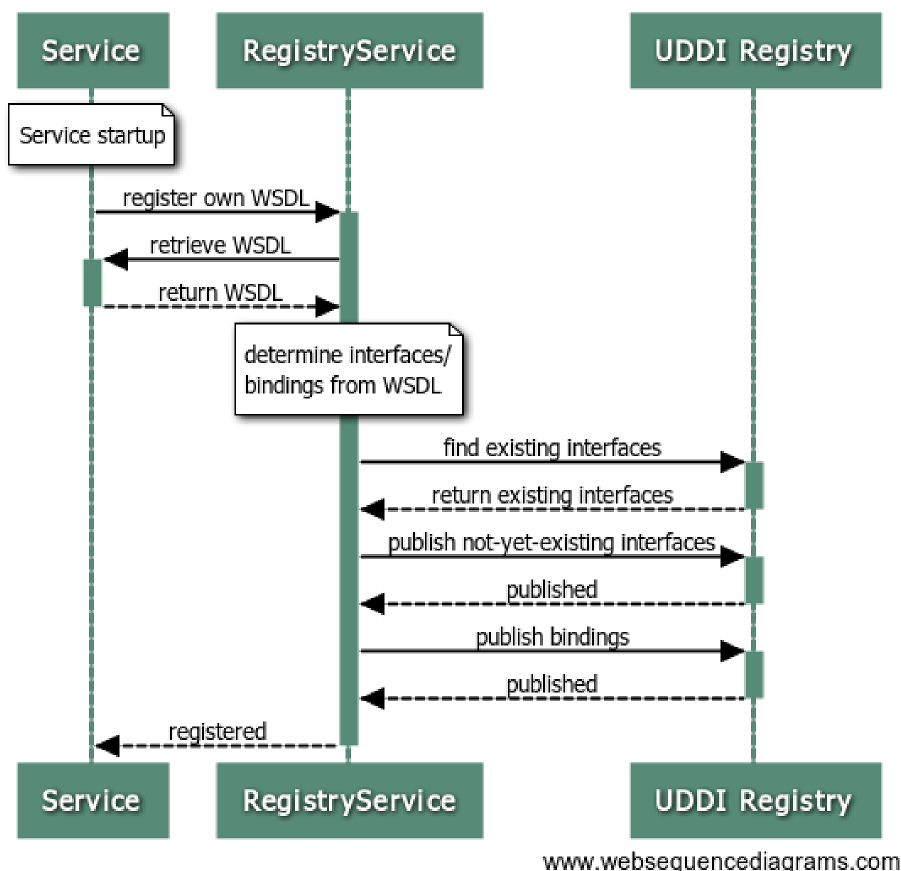


Figure 5.3: Infrastructure Interactions: Registration

When a Service starts up, it determines its own WSDL document and the Registry Service to contact, and sends a message to the Registry Service to register that WSDL location. The Registry Service in turn retrieves the WSDL document, and parses it to

determine the contained interfaces and bindings. It then publishes the information to the actual UDDI registry, making sure that all interfaces have been published before the bindings referencing them are published.

5.2.2 Workflow Planning

The use case which involves the most different components (in fact, it might use all of the components in the infrastructure) is the planning of a workflow for execution, depicted in Figure 5.4. We will shortly describe each of the interactions in the following.

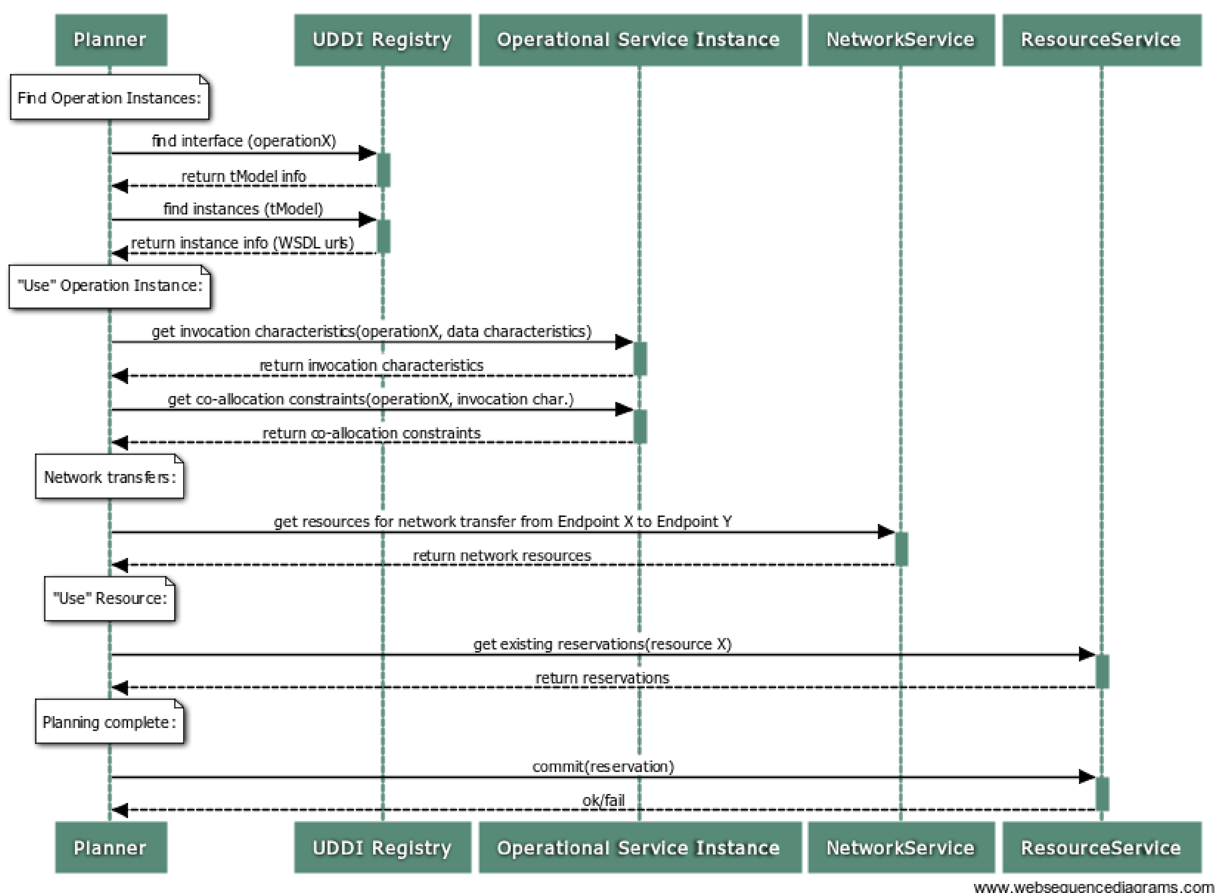


Figure 5.4: Infrastructure Interactions: Planning

- **Find Operation Instances:** for every operation contained in the workflow, it is required to determine the set of operation instances providing them. This information is obtained by querying the UDDI registry. Note that the same kind of requests is also performed to determine the required DWARFS infrastructure service instances (i.e., Workflow Engines, Storage Nodes, Network Services), but they have been omitted from the figure in order not to clutter it with too many similar details.

- **Use Operation Instance:** whenever a particular Operation Instance is considered during planning, the co-allocation constraints for the (prospective) invocation must be determined. As described in Chapter 2, this involves determining the invocation characteristics, and deriving the co-allocation constraints from them. Both of that information is obtained from the service instance being considered.

- **Network transfers:** if data is to be shipped from one endpoint (DWARFS Server) to another, the NetworkServices are contacted and queried for the resources which must be considered for that transfer.

- **Use Resource:** in order to determine feasible co-allocations which fulfil specific co-allocation constraints, one must determine all pre-existing reservations involving the required resources. This information is obtained from the ResourceServices managing the respective resources.

- **Planning complete:** once all required co-allocations have been found, they must be committed to the respective ResourceServices.

Note that, as stated earlier, Figure 5.4 is not to be strictly interpreted as a sequence diagram – the abovementioned interactions can each take place many times (with different endpoints or parameters), and in any order. However, wherever possible, the results of queries are cached, so that each individual unique query will only take place once.

5.3 Practical Considerations

5.3.1 Ramifications of using a UDDI Registry

As stated above, deployment information is stored in one or more UDDI registries. In principle, such a registry must simply be able to answer the following questions:

1. Which operations are available?

2. Where are these operations available?

As the DWARFS operations are implemented as standard WSDL/SOAP Web Services, we chose to adopt the best practices described in [B+01] for representing the required information in the UDDI registry. An overview of the mapping between WSDL and UDDI concepts is shown in Figure 5.5. While we will not go into unnecessary technical detail here, there are a few noteworthy aspects:

First, the WSDL definitions for service interfaces and service implementations must be kept separately accessible. This is because a service interface definition is a "first class citizen" (i.e., a full-fledged, standalone entity) in the UDDI registry.

Second, in practice only a single binding (and thus, port type) can be defined in a service interface WSDL document. The reason is that the UDDI tModel name is assigned from the namespace of the WSDL's top-level `<definitions>` element. If fully

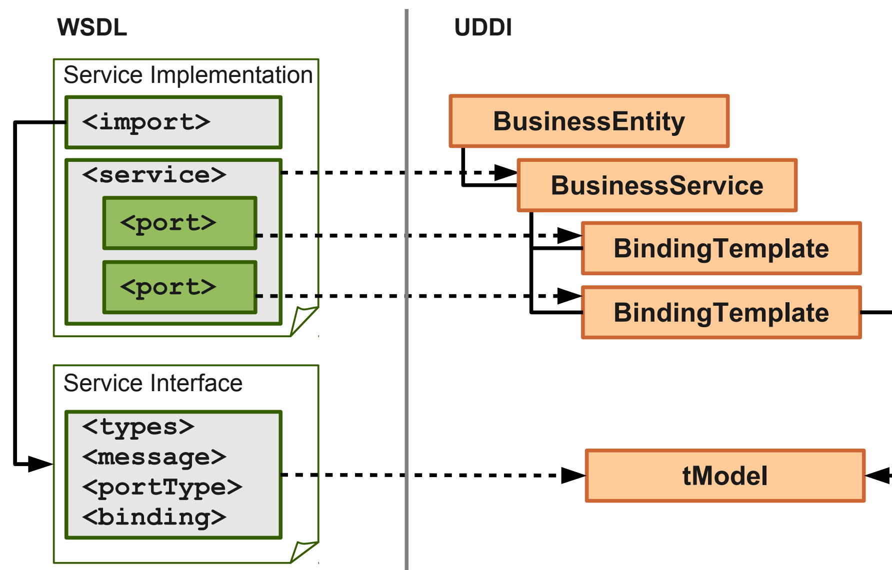


Figure 5.5: Overview of WSDL to UDDI mapping, according to [B+01]

automatic registry operation is to be supported, that namespace, in turn, is the only identifier which can be used in an implementation WSDL document to unambiguously refer to an interface document. In other words: if multiple bindings were defined in an interface WSDL document, multiple tModels with the same name would exist in the UDDI registry. While this is not forbidden, it would factually become impossible to unambiguously find the correct tModel for the implementation's BindingTemplate.

5.3.2 Glassfish Configuration and Adaptation

The Glassfish Application Server, in its default configuration, is not prepared for handling the resource-intensive and long-running types of operations that DWARFS is targeting. Therefore, the configuration must be slightly adapted to perform as expected.

First, the default network configuration for the HTTP server specifies both an upload timeout (300 seconds) and a request timeout (900 seconds). The former limits the time for which a client is allowed to send input data, while the latter is a limit for the total execution time of a single request (including the upload). Both of these timeouts should be disabled when Glassfish is to be used in a DWARFS infrastructure.

Second, the HTTP server's default thread pool configuration allows for a maximum of only 5 concurrent threads. This can quickly lead to congestion, or even deadlocks: for instance, as DWARFS services automatically contact a Registry service on startup to register themselves in the infrastructure – and the registry in turn connects back to retrieve the services' WSDL documents – the system can enter a deadlock on startup, if more than 5 services are installed. Thus, the maximum thread pool size has to be raised significantly, for instance to 500 or more.

5.3.3 Requirements for Operational Service Implementations

In principle, DWARFS can invoke arbitrary Web Service operations – ideally treating it completely as a "black box", i.e., being completely agnostic of the target services' domain-specific functionality and data. However, in practice this is not entirely feasible, for two reasons. First, the system model requires the operation instances to a) know about their resource requirements, and b) provide specific information about them, such as invocation characteristics, and co-allocation constraints. Second, while the goal was to incorporate as much enforcement complexity as possible into the DWARFS library (thus keeping it away from the Operational Service implementation), this is not entirely possible. Therefore, a Web Service meant to be deployed in DWARFS must be aware of its environment, and must interact with the DWARFS library on a few occasions.

This section describes the required adaptations. Because the topics discussed here are directly related to the implementation, and because it does not make sense to discuss some aspects without practical examples, it also contains concrete code examples where necessary. However, some irrelevant details have still been omitted.

Providing the Information required by the System Model

An Operational Service supporting DWARFS (or, "compatible service" for short) must not only provide its actual operations' functionality, but also metadata about it. In practice, this means that a compatible service must implement an additional, DWARFS-specific interface – or, in terms of our implementation: that it must provide include a port for a DWARFS-specific binding called `ResourceAwareInterface`. There are two important operations, namely `getInvocationCharacteristicsScript` and `getCoAllocationConstraintScript`. Both of them take an operation name (encoded as a QName) as input, and produce a Javascript code fragment as output. This Javascript code is then invoked at planning time with the required parameters.

Using Javascript has multiple advantages: first, it is a full-fledged programming language, so it can easily contain complex operations which might be needed to determine the result. Second, it is an interpreted language and can thus be loaded and interpreted at runtime. Third, as the base Java distribution already contains an appropriate Javascript interpreter, the scripts can directly create and operate on Java objects, and seemlessly integrate with the rest of the system. Finally – and most importantly – this approach allows to retrieve the script once, then invoke it as often as required during planning.

Registering with the UDDI service

Because of the way that Jax-WS Web Services are implemented in Glassfish, it is generally impossible to have a single class implement multiple port types. In other words, each port must be defined in its own class. While this is normally rather cumbersome, in this particular case, it can even be considered welcome: it allows the class implementing the `ResourceAwareInterface` to also perform the registration of the service, by extending the `SelfRegisteringService` class (provided by the DWARFS library).

Supporting Large Data Transfers

While this is a general Jax-WS topic rather than a DWARFS-specific one, it will apply to almost every compatible service handling large data. In order to correctly support streaming of large data, it is necessary to:

1. annotate the implementing class with the `javax.xml.ws.soap.MTOM` annotation
2. annotate the implementing class with the `com.sun.xml.internal.ws.developer.StreamingAttachment` annotation
3. ensure that streamed data is processed using `javax.activation.DataHandler`.

Supporting Resource Reservation Enforcement

In order to enable the bandwidth enforcement for a compatible service, two special configuration files must be included with the service package. These configuration files affect the Glassfish tubes configuration, i.e., the classes which perform message handling within the container, before and after the actual service invocation.

During an operation invocation, one can obtain the currently active `CoAllocation` object from the servlet request context. In order to ensure that bandwidth enforcement is performed for the response, this object must be explicitly propagated to the servlet's response context.

The object is also required for CPU enforcement. Because operation implementations (i.e., Java methods) may start to execute before all data is streamed, it is necessary to programmatically trigger the CPU enforcement (after all DataHandlers have been read).

Finally, storage enforcement must be performed by using specific classes for reading/writing files, and other resource enforcement (such as service-specific hardware) must be handled explicitly in user code.

5.3.4 Concurrent Planning and Reservations

In the current implementation, every planning run takes place individually. As shown in Figure 5.4, a planner will request information from various services available in the infrastructure. However, this information is cached locally, and multiple planners are neither aware of each other, nor are they aware of changes to the infrastructure state (caused for instance by the appearance of new Operation Instances, or by newly committed reservations). In other words, planners perceive the infrastructure state as essentially static.

This leads to a situation where the information available to planners gradually becomes stale, especially in heavily dynamic infrastructures. In the best case, this will only lead to a planner having an incomplete view of the system's state (such as not knowing that alternative Operation Instances exist), which can result in suboptimal planning results. In the worst case, a planner may try to reserve resource allocations which are no longer feasible and therefore rejected at the time of trying to commit them, thus invalidating the entire planning run. Note however that the Reservation State of the entire

infrastructure will still always remain in a valid state, specifically because it rejects commits which would lead to the overbooking of resources.

A relatively straightforward solution to address this problem is the employment of a publish/subscribe mechanism such as WS-Notification: rather than requesting required information from a DWARFS service just once, planners would subscribe to that information, so that they receive a notification whenever the respective state changes. Such a state change might still temporarily invalidate the (current) planner result. However, such a condition can easily be handled by re-calculating (only) the current GA chromosomes population given the updated infrastructure state, without having to discard or abort the planning.

Another concurrency aspect concerns the final step of the planning, namely actually committing reservations. The formal model defines this as an atomic operation corresponding to a transition from one Reservation State to another. From an implementation perspective however, that Reservation State is represented by the states of multiple independent `ResourceService` instances. Therefore, if transactional guarantees are required, one would need to employ protocols specificilly designed for distributed transaction handling. In the current implementation, we use a simple optimistic approach which tries to commit the required co-allocations sequentially at all affected resource providers.

While the current research prototype does not address the aforesaid concerns, they are of practical relevance in a production system, and can be addressed as described above.

6

Evaluation

In this chapter, we will present the evaluations that we have performed to test the effectiveness of the prototypical DWARFS implementation. As we have previously discussed, a workflow execution in DWARFS can only be performed after an initial planning, which finds and reserves the resources required during the execution. That distinction between the two phases is also reflected in the organization of this chapter. Section 6.1 presents the evaluation of the planner component, i.e., "how good are the plans that are scheduled?". Afterwards, Section 6.2 focuses on the assessment of the resource enforcement, i.e., "how well do the executions respect the reservations?". Finally, the chapter concludes with a discussion of the results.

6.1 Planner Evaluation

As described in previous chapters, our approach uses a Genetic Algorithm to plan workflow executions before they actually take place. Planning is mandatory in DWARFS, because the resources required to execute a process must be reserved in advance. Thus, the final result of planning an execution is a set of resource reservations. Of course, planning must take into account existing resource reservations, but also user-provided optimization criteria (e.g., "finish as early as possible", etc.).

To evaluate the effectiveness of our approach, we decided to simulate various scenarios. An important aspect of DWARFS is its support for *distributed* workflow execution. Therefore, one requirement for the evaluation was that the setup encompass multiple interconnected sites. In order to devise a realistic (simulated) infrastructure, we have chosen to mimic a real life setup, namely one which is based on the existing Amazon AWS infrastructure.

Table 6.1 shows the sites which are considered throughout the following evaluation. The sites are in fact a subset of the existing AWS regions (the Amazon naming was kept). The inter- and intra-site connectivities (kB/second) are approximations of actual sample measurements performed in January 2013, and the depicted costs are the official ones at that time. As can be seen, all sites offer intra-site connectivity of approximately 8.5 MB/second, which corresponds to a 100 MBit Ethernet network. Network connec-

tions between different sites expose significant differences in transmission speed, ranging between 500 kB/second (e.g., Ireland ↔ Sydney) and 2 MB/second (e.g., Singapore ↔ Tokyo). Inter-site connectivities are mostly, but not always, symmetric: for instance, we found data transfers from Virginia to Ireland to be slightly faster than those in the reverse direction. Finally, the depicted costs apply to all data transferred out of a particular site (regardless of the destination), but not to intra-site transfers.

From \ To	Ireland	Singapore	Sydney	Tokyo	Virginia	Outgoing Cost
Ireland	8500	500	500	500	1500	$ 0.12/GB
Singapore	500	8500	1000	2000	1000	$ 0.19/GB
Sydney	500	1000	8500	1500	500	$ 0.19/GB
Tokyo	500	2000	1500	8500	1000	$ 0.20/GB
Virginia	2000	500	500	1000	8500	$ 0.12/GB

Table 6.1: Sites Connectivity and Transfer Costs

The scenarios which are presented in the rest of this section focus on different aspects of the planner: The first scenario represents a realistic workflow scheduled multiple times on a large-scale infrastructure. The goal of this scenario is to the assess the overall quality of our approach by examining various characteristics of the resulting workflows, and of the infrastructure as a whole. The second scenario focuses on one particular aspect of the planning, namely the planning of data transfer strategies using Storage Nodes.

6.1.1 Simulated Deployment

The scenarios use slightly different deployments (networked sites, operation provider deployment). In the following, we will focus on the first scenario, as it is the most complex.

The workflow of this scenario requires 11 different operations (services), which shall be deployed at the different sites. In order to execute at all, Workflow Engines are also required. The first question is thus: where are operations and Workflow Engines to be deployed? In order to eliminate as much "human bias" as possible, we decided to have the infrastructure randomly created and "populated" with services. We gave the following constraints:

- Each site contains between 10 and 25 hosts.
- Each (operational) service is deployed on at least one host, and on at most half of them.
- There are at least 2, at most 10 Workflow Engines per site.

In addition, to simulate performance differences, each host was randomly assigned to one of three performance "types" (small, medium, or large). A small host has a single CPU. A medium host has two CPUs, each of which is twice as fast as a "small" CPU. A large host has four CPUs, each of which is four times as fast as a small one.

6.1.2 Scenario 1: Weather Forecast Workflow

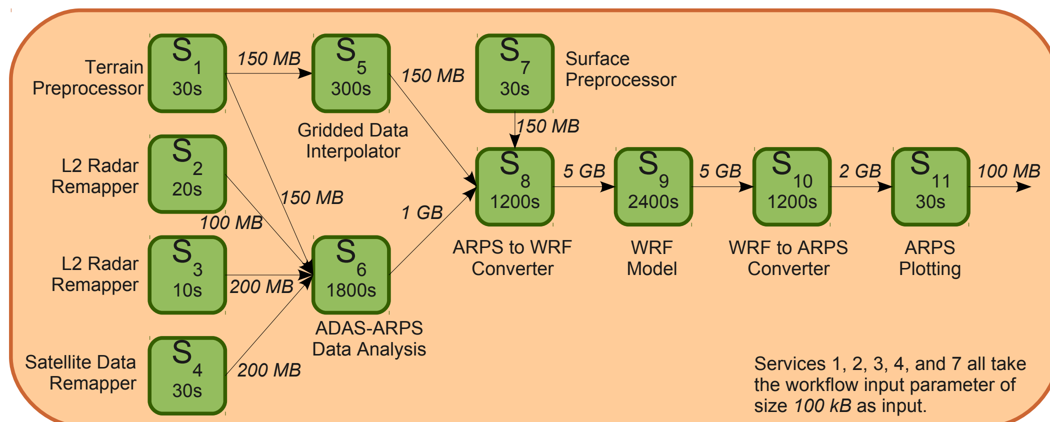


Figure 6.1: Planning Scenario 1: Process Definition and Characteristics

In this scenario, a workflow consisting of 11 operations is to be planned. The operations produce various amounts of data, and have different complexity (and thus run times). This workflow, depicted in Figure 6.1, is inspired by an actual workflow used for weather forecasts [DGR+05]. The data volumes and runtimes are estimates based on previous experiences with a workflow from a similar domain [CAA+07]. The simulated infrastructure used for this scenario is depicted in Tables 6.2 and 6.3.

Evaluation with 50 Processes

In this evaluation, the abovementioned workflow was scheduled 50 consecutive times on an initially empty infrastructure. Each execution was planned after all previous processes had committed their reservations, and the goal was always to finish as early as possible. Each optimization run lasted for 5000 generations of the Genetic Algorithm.

Figure 6.2 shows the result of those 50 planning runs. There are three immediate observations: first, the very first process planned is the one which finishes (or rather: is planned to finish) earliest. This is not surprising, given that it was scheduled on a completely empty infrastructure, while all subsequent processes have to consider the resources occupied by previous plannings. Second, there is an overall tendency for the processes planned later on to also finish later. But third, this "tendency" is not strictly monotonous, but rather, some schedules are significantly faster to terminate than others, which had previously been scheduled. We will now discuss these seemingly strange results.

The results are caused by a combination of different properties, namely the characteristics of the workflow, the infrastructure, and the employed metaheuristic itself. Before we go into further details, let us shortly revise these.

The very reason for employing a Genetic Algorithm is that it is generally impossible to analytically determine an optimal solution because of the prohibitive effort. In this particular scenario, there are 27 Workflow Engines and 11 Operations, each of which is provided by between 15 and 31 providers. In principle, every operation can

Hostname	type	WFE	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
ireland01	L				x	x	x					x	x
ireland02	S							x				x	x
ireland03	S	x			x						x		
ireland04	M	x				x	x		x	x	x		x
ireland05	S				x	x		x	x		x		
ireland06	M					x		x			x	x	
ireland07	M		x	x	x	x	x						x
ireland08	M	x											x
ireland09	M	x			x		x						
ireland10	L				x			x	x				x
ireland11	M	x			x	x				x	x	x	
ireland12	L	x		x			x		x		x	x	
ireland13	L	x		x			x	x		x			x
ireland14	M	x							x		x	x	
singapore01	S					x							
singapore02	M	x				x			x		x	x	
singapore03	L			x									
singapore04	M	x										x	x
singapore05	M		x										
singapore06	S	x			x	x	x					x	
singapore07	S	x	x		x					x		x	
singapore08	S				x					x		x	
singapore09	L		x								x		
singapore10	M										x		
singapore11	L		x	x				x				x	
singapore12	S	x			x	x							
singapore13	M				x						x	x	
singapore14	M												x
sydney01	L												
sydney02	L						x						
sydney03	S			x	x								
sydney04	S	x	x		x	x	x			x			
sydney05	M		x		x							x	
sydney06	S					x	x				x	x	
sydney07	S			x	x	x				x			x
sydney08	L			x		x				x			
sydney09	M		x		x	x			x	x		x	
sydney10	S	x	x		x		x		x				x
sydney11	S		x						x				
sydney12	M			x		x		x	x	x			
sydney13	L		x				x				x		
sydney14	S		x	x		x	x		x			x	
sydney15	M				x								
sydney16	S					x					x	x	
sydney17	M					x							
sydney18	L		x		x							x	
sydney19	M		x		x								
Hostname	type	WFE	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}

Table 6.2: Planning Scenario 1: Infrastructure Deployment

Hostname	type	WFE	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
tokyo01	M							x					x
tokyo02	L	x			x	x	x		x	x	x		
tokyo03	L	x						x				x	
tokyo04	L	x	x			x	x	x	x				
tokyo05	M	x											
tokyo06	S		x		x				x				
tokyo07	S		x		x								
tokyo08	S	x				x		x		x			
tokyo09	L	x			x			x					
tokyo10	S	x		x	x			x					
tokyo11	L	x	x										
tokyo12	M				x					x			
virginia01	M	x			x			x	x	x			
virginia02	L	x											x
virginia03	L		x				x			x			
virginia04	L					x							
virginia05	L							x					
virginia06	M				x		x			x			x
virginia07	M			x							x		
virginia08	M			x	x								
virginia09	L									x	x		
virginia10	L					x	x	x					
virginia11	S					x		x			x	x	
virginia12	S					x			x		x		
virginia13	S	x		x				x					
virginia14	L					x	x		x				
virginia15	S			x		x			x				
virginia16	L		x			x		x			x	x	x
virginia17	M	x			x	x		x			x		
virginia18	S					x		x					
Hostname	type	WFE	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}

Table 6.3: Planning Scenario 1: Infrastructure Deployment (contd.)

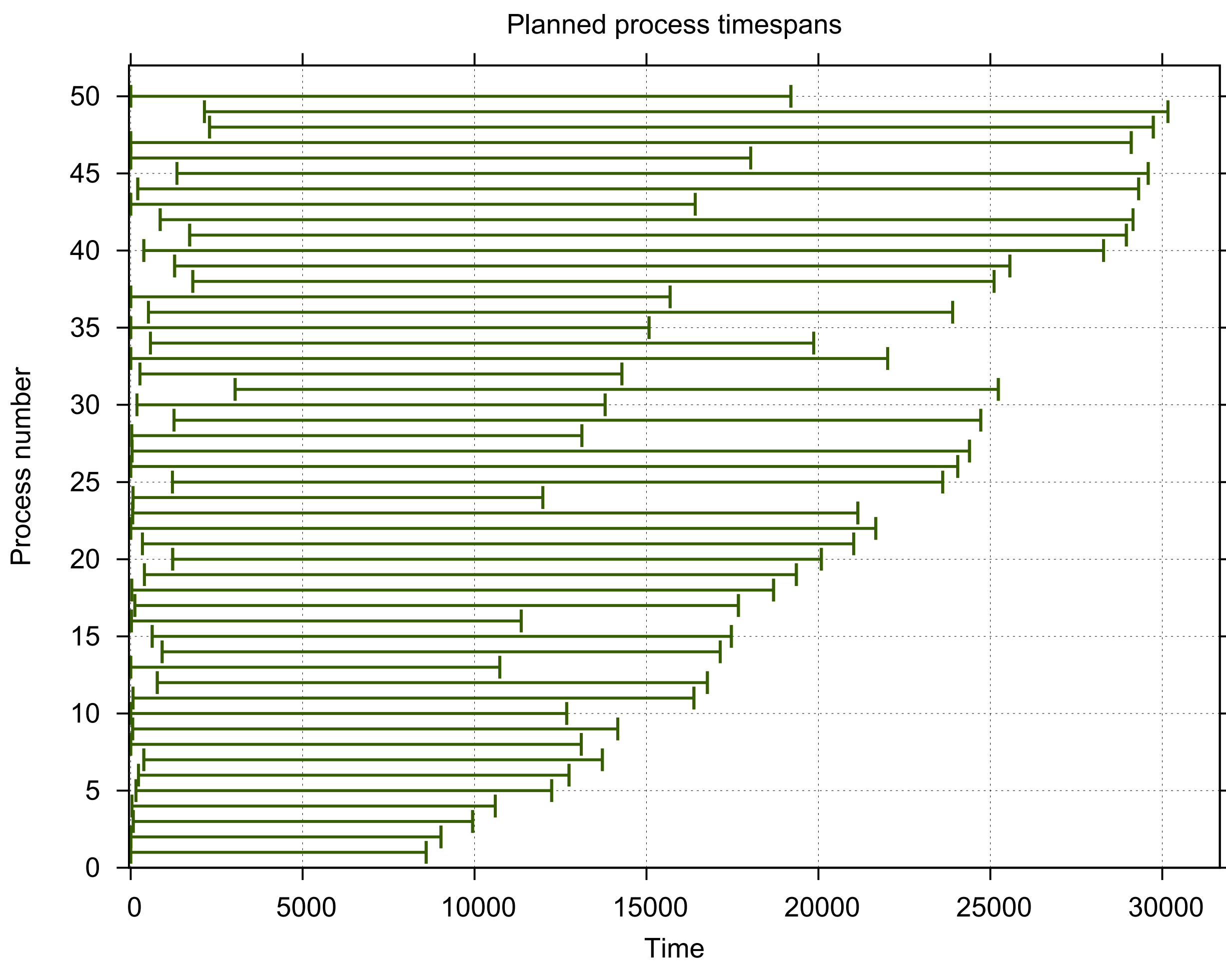


Figure 6.2: Planning Scenario 1, 50 Processes: Planned Runtimes

be performed anywhere, and by any Workflow Engine, which results in approximately 3.7×10^{36} different combinations. In addition, the minimum and maximum usages for each required resource can be mutated (in this setup, in steps of 1%), which yields another factor of 10^{102} for this particular workflow. Especially if there already exist resource reservations in the infrastructure at planning time, it is generally not possible to determine which of these combinations perform better than others (in terms of the planning goal). The task of the metaheuristic is thus to find a solution which is "as good as possible" – but one cannot expect the found solution to be the absolute optimum.

Let us now consider the workflow definition, and the deployment present in the infrastructure. The most resource-intensive part of the workflow – in terms of both operation duration and data size – is the fragment encompassing Operations 6, 8, 9, and 10. Thus, if the goal is fast execution, one would intuitively want to "co-locate" their execution to a single site in order to speed up data transfers. One would also want to execute those expensive operations on fast machines (hosts of type `L`). Tables 6.4 and 6.5 show the allocations that were planned for the first 16 processes for these activities.

Process	S_6 (upld.)	S_6 (exec.)	S_6 (dnld.)	S_6-S_8 (transf.)	S_8 (upld.)	S_8 (exec.)	S_8 (dnld.)	S_8-S_9 (transf.)	S_9 (upld.)	S_9 (exec.)	S_9 (dnld.)	S_9-S_{10} (transf.)	S_{10} (upld.)	S_{10} (exec.)	S_{10} (dnld.)
Process 1	WFE:		ireland13		WFE:		ireland13		WFE:		ireland12		WFE:		ireland12
	OP:		ireland13		OP:		ireland13		OP:		ireland12		OP:		ireland12
start	116	130	1930	1961	1983	2011	3211	3342	3974	4082	6482	6590	6698	6806	8006
duration	14	1800	31	22	28	1200	131	632	108	2400	108	108	108	1200	43
max. usage	100%	25%	70%	100%	100%	25%	82%	100%	100%	25%	100%	100%	100%	25%	100%
Process 2	WFE:		ireland13		WFE:		ireland13		WFE:		ireland12		WFE:		ireland12
	OP:		ireland13		OP:		ireland13		OP:		ireland12		OP:		ireland12
start	441	457	2257	2299	2385	2413	3690	3974	4606	4714	7114	7222	7330	7438	8638
duration	16	1800	42	86	28	1277	215	632	108	2400	108	108	108	1200	43
max. usage	88%	25%	52%	25%	100%	23.5%	50%	100%	100%	25%	100%	100%	100%	25%	100%
Process 3	WFE:		virginia13		WFE:		ireland13		WFE:		ireland12		WFE:		ireland12
	OP:		virginia10		OP:		ireland13		OP:		ireland12		OP:		ireland12
start	487	568	2368	2495	3120	3187	4464	4606	5238	5346	7746	7854	8049	8157	9357
duration	81	1800	127	625	67	1277	128	632	108	2400	108	115	108	1200	43
max. usage	100%	25%	100%	86%	42%	23.5%	84%	100%	100%	25%	100%	94%	100%	25%	100%
Process 4	WFE:		virginia02		WFE:		ireland13		WFE:		ireland12		WFE:		ireland12
	OP:		virginia16		OP:		ireland13		OP:		ireland12		OP:		ireland12
start	587	692	3000	3141	3678	3825	5025	5238	5870	5978	8378	8486	8681	8792	9992
duration	105	2308	141	537	147	1200	131	632	108	2400	108	108	111	1200	70
max. usage	77%	19.5%	90%	100%	19%	25%	82%	100%	100%	25%	100%	100%	97%	25%	62%
Process 5	WFE:		ireland04		WFE:		ireland04		WFE:		ireland12		WFE:		ireland12
	OP:		ireland10		OP:		ireland04		OP:		ireland12		OP:		ireland12
start	752	833	3691	4033	4080	4108	6508	6664	7638	7746	10146	10254	10362	10470	11670
duration	81	2858	342	47	28	2400	156	821	108	2400	108	108	108	1200	68
max. usage	100%	15.75%	37%	46%	100%	50%	69%	77%	100%	25%	100%	100%	100%	25%	64%
Process 6	WFE:		ireland04		WFE:		ireland04		WFE:		ireland12		WFE:		ireland12
	OP:		ireland10		OP:		ireland13		OP:		ireland12		OP:		ireland12
start	833	1450	3250	4830	4864	5185	6631	7560	8257	8378	10778	10886	10994	11102	12302
duration	617	1800	1580	34	321	1446	929	632	121	2400	108	108	108	1200	43
max. usage	13%	25%	8%	64%	51%	20.75%	68%	100%	89%	25%	100%	100%	100%	25%	100%
Process 7	WFE:		ireland08		WFE:		ireland11		WFE:		ireland12		WFE:		ireland12
	OP:		ireland10		OP:		ireland13		OP:		ireland12		OP:		ireland12
start	864	1229	3298	3640	3883	4057	6126	8192	9237	9357	11757	11865	11973	12081	13281
duration	365	2069	342	243	174	2069	1707	810	120	2400	108	108	108	1200	80
max. usage	22%	21.75%	37%	52%	94%	14.5%	43%	78%	90%	25%	100%	100%	100%	25%	54%
Process 8	WFE:		ireland13		WFE:		ireland04		WFE:		ireland04		WFE:		ireland12
	OP:		ireland13		OP:		ireland04		OP:		ireland04		OP:		ireland12
start	383	465	2265	2306	2477	2505	4905	5134	5242	5350	10150	10508	11140	11273	12473
duration	82	1800	41	171	28	2400	229	108	108	4800	358	632	133	1200	63
max. usage	18%	25%	75%	74%	100%	50%	47%	100%	100%	50%	30%	100%	81%	25%	69%

Table 6.4: Allocations for Process Fragment (Processes 1 - 8)

Process	S_6 (upld.)	S_6 (exec.)	S_6 (dnld.)	S_6-S_8 (transf.)	S_8 (upld.)	S_8 (exec.)	S_8 (dnld.)	S_8-S_9 (transf.)	S_9 (upld.)	S_9 (exec.)	S_9 (dnld.)	S_9-S_{10} (transf.)	S_{10} (upld.)	S_{10} (exec.)	S_{10} (dnld.)
Process 9	WFE: virginia01				WFE: ireland11				WFE: ireland11				WFE: ireland12		
	OP: virginia05				OP: ireland11				OP: ireland11				OP: ireland12		
start	1078	1183	2983	3110	3647	3675	6075	6183	6466	6582	11382	11536	12168	12282	13482
duration	105	1800	127	537	28	2400	108	283	116	4800	154	632	114	1200	94
max. usage	77%	25%	100%	100%	100%	50%	100%	38%	93%	50%	70%	100%	95%	25%	46%
Process 10	WFE: ireland11				WFE: ireland11				WFE: ireland11				WFE: ireland11		
	OP: ireland10				OP: ireland11				OP: ireland11				OP: ireland01		
start	172	253	2053	2180	2202	2230	4630	4754	4862	4970	9770	9878	9986	10618	11818
duration	81	1800	127	22	28	2400	124	108	108	4800	108	108	632	1200	253
max. usage	100%	25%	100%	100%	100%	50%	87%	100%	100%	50%	100%	100%	100%	25%	100%
Process 11	WFE: ireland08				WFE: ireland14				WFE: ireland14				WFE: ireland12		
	OP: ireland10				OP: ireland13				OP: ireland14				OP: ireland12		
start	833	1101	4083	4238	4753	5337	7560	8534	8711	8945	13745	13955	14587	14701	15901
duration	268	2982	155	201	584	2223	974	177	234	4800	114	632	114	1200	79
max. usage	30%	15.75%	82%	63%	28%	13.5%	68%	61%	46%	50%	95%	100%	95%	25%	55%
Process 12	WFE: virginia13				WFE: ireland13				WFE: ireland12				WFE: ireland12		
	OP: virginia05				OP: ireland13				OP: ireland12				OP: ireland12		
start	3449	3530	5330	5564	6142	6172	8019	9023	12345	12453	14853	14961	15069	15177	16377
duration	81	1800	234	578	30	1847	127	1109	108	2400	108	108	108	1200	43
max. usage	100%	25%	54%	93%	93%	16.25%	85%	57%	100%	25%	100%	100%	100%	25%	100%
Process 13	WFE: virginia17				WFE: virginia17				WFE: virginia17				WFE: virginia17		
	OP: virginia16				OP: virginia09				OP: virginia09				OP: virginia16		
start	241	374	2174	2301	2323	2487	3687	4319	4427	5059	7459	8091	8199	8831	10031
duration	133	1800	127	22	164	1200	632	108	632	2400	632	108	632	1200	253
max. usage	83%	25%	100%	100%	100%	25%	100%	100%	100%	25%	100%	100%	100%	25%	100%
Process 14	WFE: ireland12				WFE: ireland04				WFE: ireland04				WFE: ireland12		
	OP: ireland06				OP: ireland04				OP: ireland04				OP: ireland12		
start	2175	2293	5893	6085	6480	6508	8908	9164	9300	9408	14208	14587	15219	15327	16527
duration	118	3600	192	342	28	2400	256	136	108	4800	108	632	108	1200	55
max. usage	68%	50%	66%	37%	100%	50%	42%	79%	100%	50%	100%	100%	100%	25%	79%
Process 15	WFE: ireland13				WFE: ireland13				WFE: ireland04				WFE: ireland04		
	OP: ireland10				OP: ireland13				OP: ireland04				OP: ireland12		
start	2786	2979	5348	5559	5618	5927	7692	7907	10033	10150	14950	15106	15219	15851	17051
duration	193	2369	211	59	309	1765	215	1071	117	4800	156	108	632	1200	253
max. usage	43%	19%	60%	37%	9%	17%	50%	100%	92%	50%	69%	100%	100%	25%	100%
Process 16	WFE: virginia01				WFE: virginia01				WFE: virginia01				WFE: virginia01		
	OP: virginia05				OP: virginia03				OP: virginia16				OP: virginia16		
start	424	511	2311	2438	2466	2630	3830	4462	4577	5443	7918	8679	8831	9463	10663
duration	87	1800	127	28	164	1200	632	115	866	2475	761	131	632	1200	253
max. usage	100%	25%	100%	77%	100%	25%	100%	94%	73%	24.25%	83%	82%	100%	25%	100%

Table 6.5: Allocations for Process Fragment (Processes 9 - 16)

As can be seen in these tables, almost all of the plannings tended to concentrate those executions in the Ireland site. Looking at the deployment, that indeed makes sense: all of the operations are available there on hosts of type `L` (`ireland12` and `ireland13`), both of which also provide a Workflow Engine, thus further reducing data transfer time.

Figure 6.3 provides further details on the resource usages on these two hosts. Let us focus on the most evident part of those figures first: All of the four first processes planned to orchestrate Operations 9 and 10 completely on `ireland12` (i.e., both the Workflow Engine and Operation Instance were chosen on that host). One can clearly see how the incoming bandwidth is used by all four processes consecutively (starting at timestamp 3342), then the CPU is used for Operation 9, followed by a short period of using the loopback (local) bandwidth, followed by the execution of Operation 10.

So while concentrating the execution of those four important operations in Ireland makes sense initially, the resources there get loaded, and eventually booked out. Put bluntly, the Genetic Algorithm "realizes" this at some point in time – for the first time in process 13, where the entire process fragment is scheduled for execution in Virginia. The same pattern can be observed in Process 16, and subsequently.

The question now is: why doesn't this happen earlier (e.g., why isn't the solution of Process 13 already found in Process 5)? We cannot fully explain this, but an educated guess would be that the heuristic gets "stuck" in a local optimum. After all, once a certain combination of workflow engines and operation providers (all of which are represented by individual, independent, genes) is found to be preferrable to other combinations, all of those genes would need to be changed in order to "switch" to a different constellation. Such a switch is eased by the (presumed) best combination becoming booked out. Figure 6.4 shows how two different planning runs (Process 12 and 13) evolved over time. Process 12 presents a common pattern, which is observed for many schedules: initially, large improvements are made, then the rate of improvement gradually attenuates, until it eventually stops – which yields a hyperbolic appearance. Process 13 initially performs similarly, however in that case, a significant improvement (namely the switch from Ireland to Virginia) occurs after around 1300 generations.

Unfortunately, because of the randomized nature of the algorithm, it is impossible to predict when – if ever – such jumps will occur. Ultimately, it is always a trade-off between spending significantly more resources to (possibly, eventually) find a better solution, and getting acceptable, though not optimal, results.

Another observation is that even if resources are available, they are not always used to the maximum possible extent. For instance, process 4 does not use 100% of the available capacity when downloading from Operation 10, but merely 62%, thus wasting 27 seconds on that particular workflow step. As Figure 6.3 demonstrates, such behavior can be observed for other processes, resources, and workflow steps.

The explanation is rather simple, but also somewhat unsatisfying: because of the enormous search space, a mutation which would optimize that particular step has simply not been tried on an individual where it would improve the overall result. However, note that not all occurences where resources aren't fully booked necessarily worsen the overall result. A prominent example in this scenario is the execution of Operation 7, and the subsequent data transfer: because that operation is short-running and produces

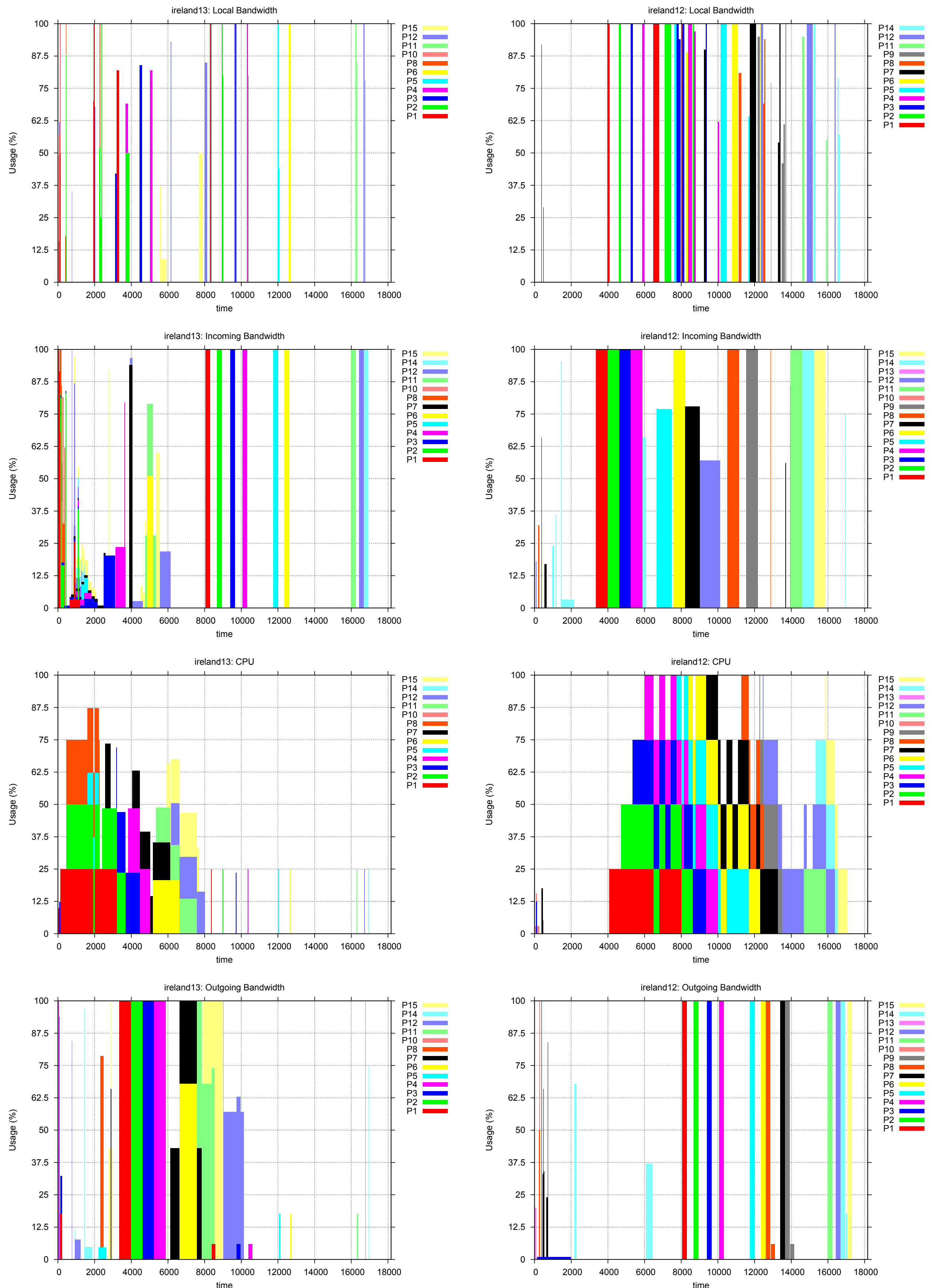


Figure 6.3: Planning Scenario 1: Resource Usages of ireland13 and ireland12

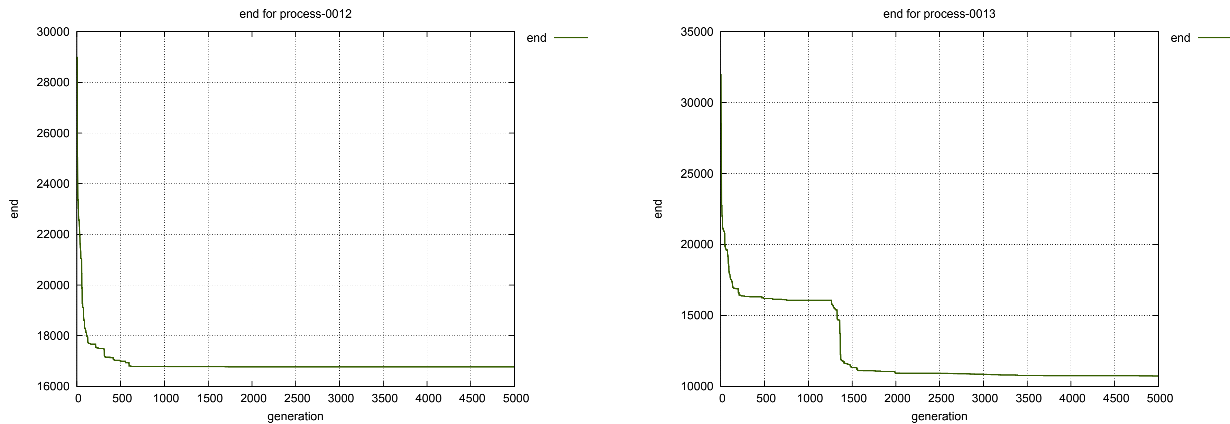


Figure 6.4: Planning Scenario 1: Evolution of the Planning of Processes 12 and 13

relatively little data, it is not on a critical path for that process – so it can very well use few resources, yet still be ready "in time".

Evaluation with 500 Processes

While the abovementioned results provide valuable first insights, they also raised an interesting aspect. In particular, the most resource-intensive part (operations 6 - 10) of almost all processes tended to get scheduled at sites with a "favorable deployment" (in terms of availability of services and WFEs, and host performace – namely Ireland and Virginia), thus disproportionally loading these sites while avoiding others. We have therefore repeated the evaluation, but with 500 consecutively scheduled processes (with 2500 GA generations each). In addition, we have also introduced parallel computations: in this case, each process was in fact planned 12 (independent) times in parallel, and the best result was chosen. The resulting planned process runtimes are shown in Figure 6.5, and discussed in the following.

A relatively good indicator of how often resources of individual sites are used can be obtained by simply considering the intra-site network usages. Figure 6.6 shows the network usages of 4 of the 5 considered networks.[1] As can be seen, all of these networks are used eventually. Ireland and Virginia are essentially booked all the time, while the "unfavorably deployed" Singapore and Sydney sites are only considered later in the planning phase (towards the "back" of the graphs). Basically, these latter sites are used as alternatives once the former (preferred) ones are too booked out.

Roughly the same pattern can be seen in Figure 6.7, which presents the resource reservations for a few representative hosts. virginia09 is one of the preferred (large) hosts for executing the resource-intensive operations 8 and 9, so it is normal for this host to be well booked. singapore11 is a (large) host providing operations 6 and 10 (in fact, the only host at all in Singapore to provide operation 6, and the only L instance for operation 10) – thus, it is also used repeatedly, but less frequently, and only in later

[1]It is normal for these graphs to show more than 100% usage, because for this particular setup, networks have been set up to allow for a maximum of 1,000,000 concurrent connections each.

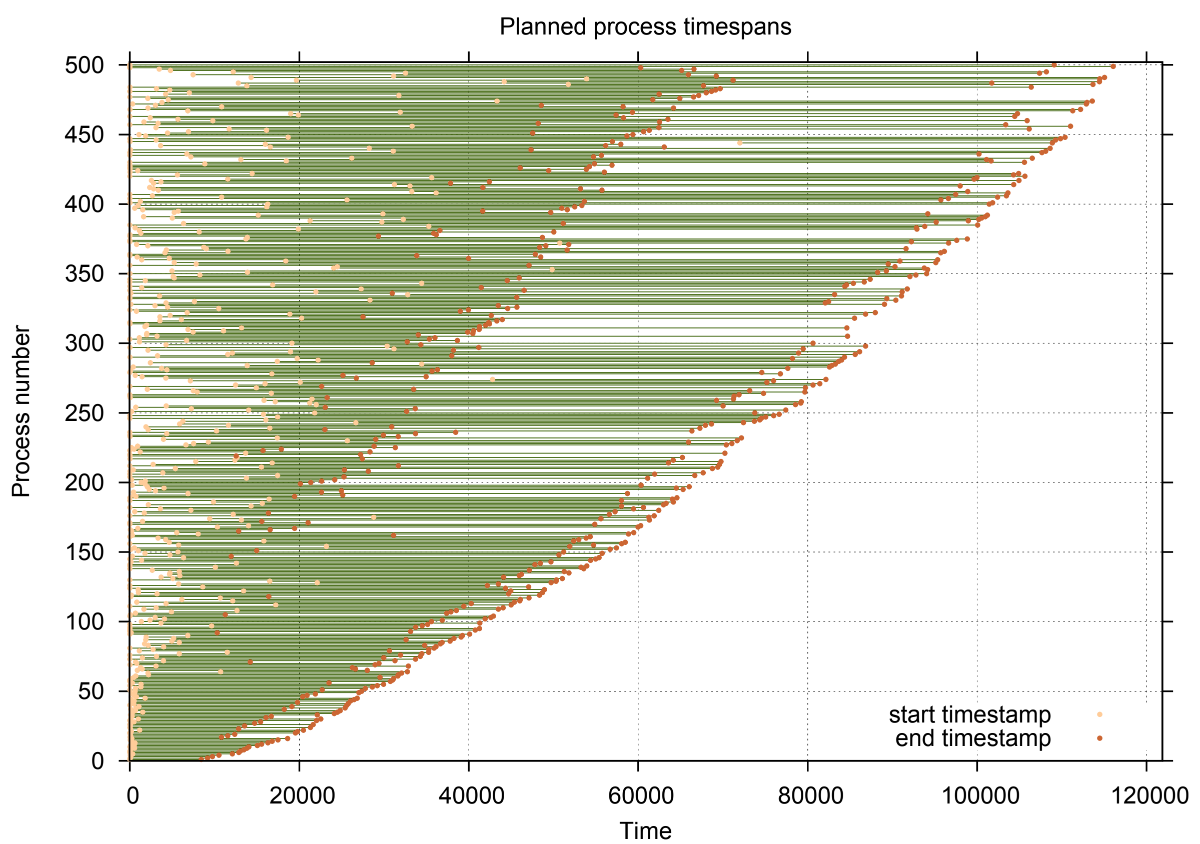


Figure 6.5: Planning Scenario 1, 500 Processes: Planned Runtimes

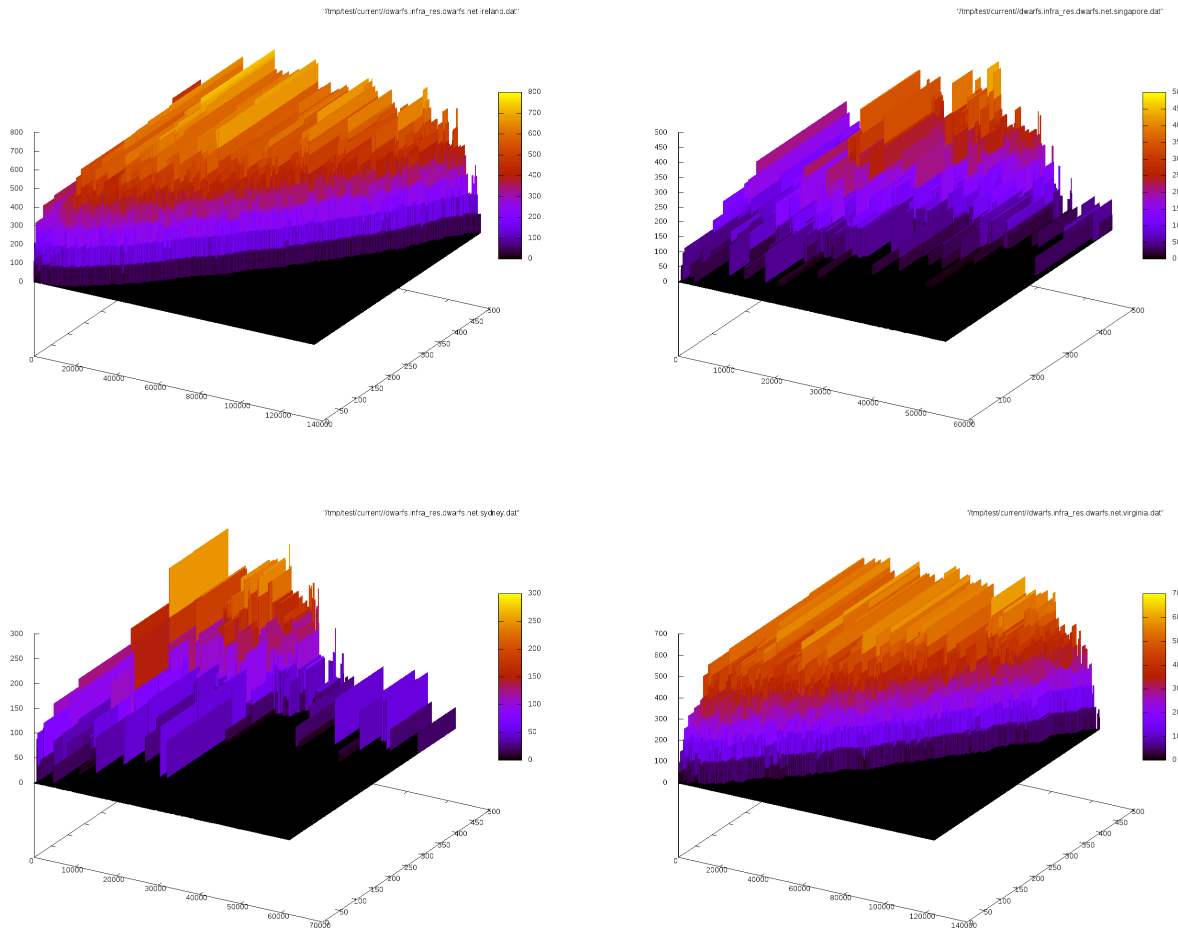


Figure 6.6: Planning Scenario 1, 500 Processes: Site Network Usages

planning runs. ireland03 shows an interesting case of a very regular "switch" to a small host providing operation 9 once the preferred large hosts are booked out.

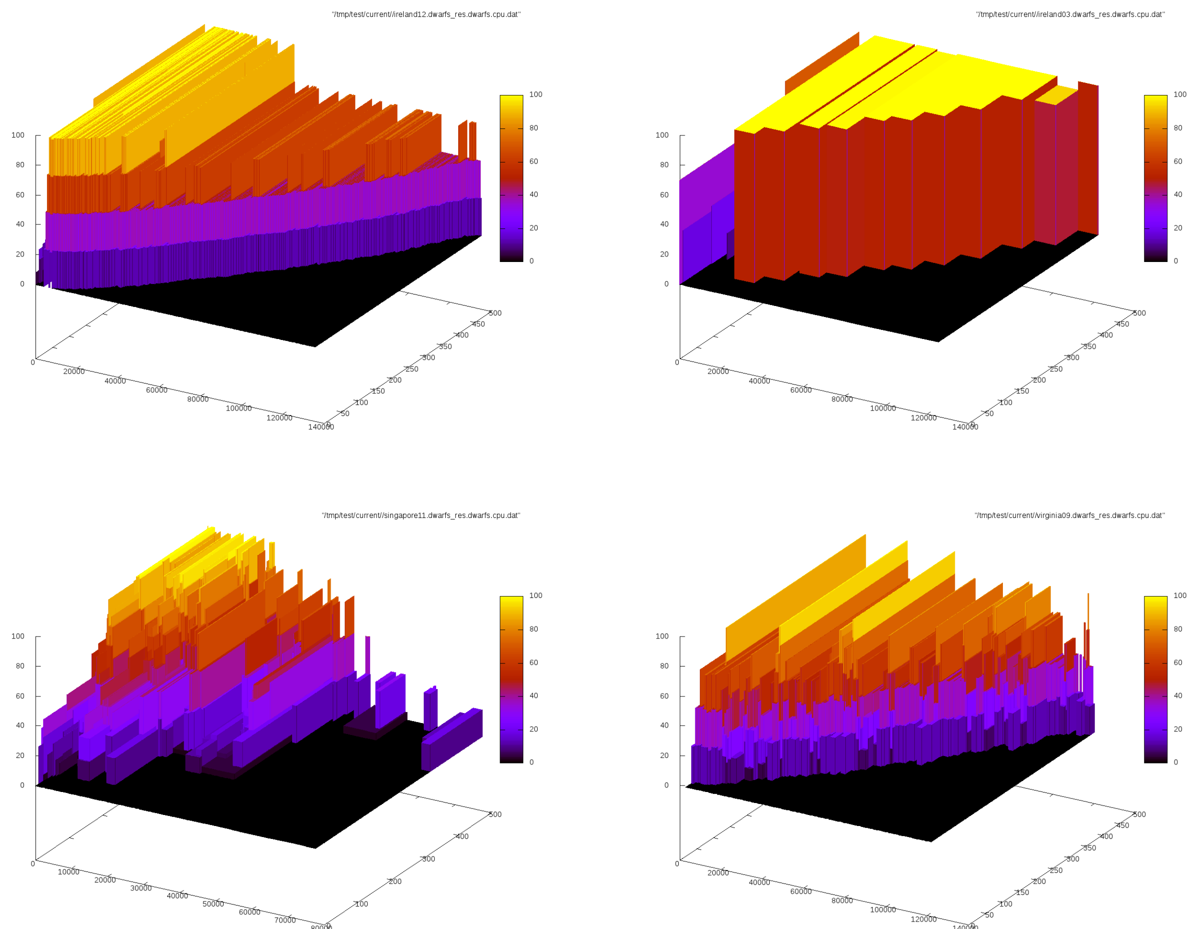


Figure 6.7: Planning Scenario 1, 500 Processes: Select Host CPU Usages

Emergence of Classes of Preferred Execution Sites

One striking observation, which can be seen very clearly in Figure 6.5, is that there are in fact four different classes of execution durations, which manifest themselves in distinct (and almost linear) "sub-curves". We have verified these to relate to the network site where the most resource-intensive part of the process is executed. In particular, the "rightmost" curve (the one producing the longest runtimes) corresponds to the Ireland site; the second one to the Virginia site. The third one, which starts to become apparent after around 150 processes have been scheduled, corresponds to the Tokyo site. Finally, the fourth class encompasses all other combinations (e.g., using combinations at different sites) – in the figure, these are the occasional outliers which don't align with any of the other curves.

Effect of Parallel Planning

Such irregular behavior – a subsequent schedule producing significantly better results than a previous one – is in fact unwanted. Ideally, one would always want to find the best combination (for instance considering a co-location at the Tokyo site in a much earlier planning run than what the evaluation produced). This was the reason why we introduced parallel planning – in this case, each planning run being performed 12 times in parallel, in the hope of having a twelve times higher chance to find the better combination. Unfortunately, the approach did not yield the improvements that we hoped for: while the results did produce slightly better results (on average, the end timestamp of the best schedule was around 5% sooner than only scheduling using a single thread), the approach did not help to mitigate the observed irregularities.

Conclusion

We believe these irregularities to be the result of a combination of different factors, namely a) the mere extent of the problem space (approximately 3.7×10^{138} possible combinations for this scenario), b) the deployment characteristics, and c) existing reservations. In particular, the deployment characteristics (i.e., which operations are deployed where, and where are workflow engines available) seem to play an important role. Put simply: the probability of randomly choosing a competitive configuration at the Tokyo site is much smaller than it is at the Ireland (or Virginia) site. Only once the latter sites are significantly booked does the "pressure" to find alternatives increase – but even then, the algorithm does not necessarily find the better alternative, because finding it requires multiple genes at once to be modified in order to perform the switch (only switching one gene, for instance attempting to invoke an operation in Tokyo, but from a Workflow Engine in Ireland, is likely to worsen the result instead of improving it). We have verified this assumption by changing the algorithm to force an initial co-location of the resource-intensive part of the workflow at the Tokyo site, and as expected, the schedules stayed there instead of switching to other sites (because again, only an all-or-none change in multiple genes would have improved the overall result, while independent single mutations would have worsened it).

In summary, we believe those irregularities in scheduling to be mostly influenced by the non-uniform deployment of services, causing different probabilities of finding, and trying, the various constellations. Conversely, a more regular deployment is likely to result in more uniform schedules.

6.1.3 Scenario 2: Data Transfer Strategies

The goal of evaluating this scenario is slightly different from the first one – here, we want to verify how well the planning performs in finding the best strategy to transfer data. The workflow used in this evaluation is depicted in Figure 6.8, and the corresponding infrastructure is shown in Table 6.6.

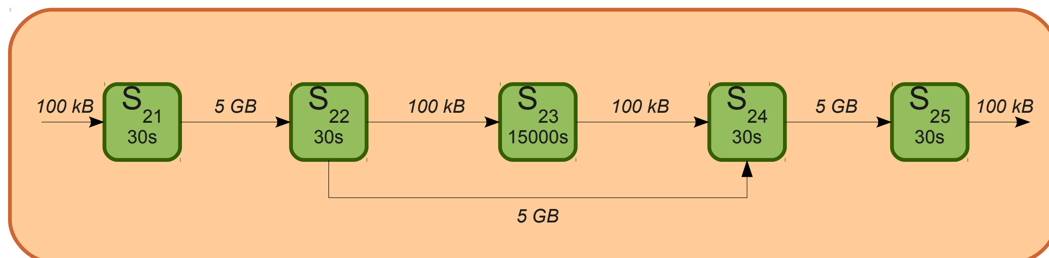


Figure 6.8: Planning Scenario 2: Process Definition and Characteristics

Hostname	type	WFE	SN	S_{21}	S_{22}	S_{23}	S_{24}	S_{25}
ireland01	M	x		x	x	x		
ireland02	S		x					
ireland03	S		x					
singapore01	S	x						
singapore02	M		x					
singapore03	L		x					
sydney01	L	x					x	x
sydney02	L		x					
sydney03	S		x					
tokyo01	M	x						
tokyo02	L		x					
tokyo03	L		x					
virginia01	M	x						
virginia02	L		x					
virginia03	L		x					
Hostname	type	WFE	SN	S_{21}	S_{22}	S_{23}	S_{24}	S_{25}

Table 6.6: Planning Scenario 2: Infrastructure Deployment

Setting

As described in Section 3.5, Storage Nodes (SN) can serve to temporarily store data during process execution, and forward them to their destination Workflow Engine when they are required. The rationale is that while data must – of course – pass through a WFE when the corresponding operation is invoked, it is not desirable to waste WFE resources by keeping large volumes of data without using them.[2] This approach may make sense in workflows where data is "produced early, but consumed late" – i.e., where the data must be preserved over longer periods of time. The workflow shown in Figure 6.8 exhibits this pattern: 5 GB of data is produced by S_{22}, and consumed by S_{24}, with a long-running activity (that does not require said data) in between. Note that this workflow (and the deployment) have been manually crafted so that they expose the following characteristics:

1. Because of the data volumes involved and the services' deployment, the only sensible strategy for orchestrating this workflow is to split its execution between two sites, i.e., to invoke S_{21} and S_{22} from the WFE at the Ireland site, and S_{24} and S_{25} from the one at the Sydney site (any other invocation strategy will result in both higher cost and longer execution time). These two sites have been chosen on purpose, because they are poorly connected.

2. Given the deployment and workflow definition, it is possible to analytically determine the fastest possible execution time. On an infrastructure with no existing resource reservations, this duration is 15994 seconds.

3. The duration for the S_{23} operation was not chosen arbitrarily, but so that it is long enough to allow for any data transfer strategy (with sufficiently high throughput) to finish in time.

4. The deployments at the other sites (Singapore, Tokyo, Virginia) have only been added to verify that they are in fact *not* used (i.e., possibly considered, but never chosen, by the planner). As mentioned above, using services at any of these sites would negatively affect both the cost and execution time.

There is one further aspect to consider, namely: what is a sensible optimization goal, i.e., which fitness function should be employed for the optimization? If one was only considering execution time (early termination), the transfer strategy would in fact be irrelevant, because any strategy can achieve the optimal duration. As the only other metric available is execution cost, we need to employ a fitness function that allows to optimize for both execution time and execution cost. We chose a fitness function which considers both goals with equal weight. The final question now is: under which circumstances will a transfer strategy that employs Storage Nodes be better (i.e., cheaper) than one which doesn't? As the incentive is to favor storing data on SNs (rather than on WFEs), it is logical that a WFE must charge more for keeping the data than a SN does. The most cost-effective strategy is then to a) quickly write the data produced by

[2] Such "waste" of WFE resources is only modelled indirectly, by affecting the reservation cost: A WFE charges considerably more than a SN for (temporarily) storing data.

S_{22} from the invoking SN to a nearby SN, b) transfer the data to a SN at the remote site, and c) quickly transfer it from that SN to the WFE invoking S_{24} just in time, before it is needed.

For brevity, we will call the two involved sites (Ireland and Sydney) A and B in the following. Let N_{AB} denote the network speed between sites A and B, in bytes per second (in this case 500,000). Let N_A and N_B denote the respective intra-site network speed (in this case both 8,500,000). Let C_A, C_B, and C_{AB} denote network cost (in cents per byte) at the respective site and between sites. Let S_A, S_B, W_A, W_B denote the cost of storing data on a Storage Node/Workflow Engine at the respective site, in cents per byte per second. Finally, Let D denote the data size (in this case 5 GB).

Note that in the following, we are only considering the minimum costs for transfers (i.e., where data is only transiting through a SN, being forwarded immediately after it is uploaded, and where all transfers occur at maximum network speed).

Direct Transfer: The minimum cost $C_{dir}(A, B)$ of a Direct data transfer from a WFE at site A to a WFE at site B is:

$$C_{dir}(A,B) = \underbrace{\frac{D}{N_{AB}}}_{\text{transfer duration}} \times \underbrace{(W_A + W_B) \times D}_{\text{WFE cost per second}} + \underbrace{C_{AB} \times D}_{\text{transfer cost}}$$

Single-Indirect Transfer: The minimum cost $C_{ind1}(A, B, A)$ of a Single-indirect data transfer from a WFE at site A to a WFE at site B, via a SN at site A, is:

$$C_{ind1}(A,B,A) = \underbrace{\frac{D}{N_A} \times (W_A + S_A) \times D}_{\text{WFE(A) and SN(A)}} + \underbrace{(C_A + C_{AB}) \times D}_{\text{transfers}} + \underbrace{\frac{D}{N_{AB}} \times (S_A + W_B) \times D}_{\text{SN(A) and WFE(B)}}$$

Double-Indirect Transfer: The minimum cost $C_{ind2}(A, B, A, B)$ of a Double-indirect data transfer from a WFE at site A to a WFE at site B, via SNs at site A then B, is:

$$\begin{aligned} C_{ind2}(A,B,A,B) = & \underbrace{\frac{D}{N_A} \times (W_A + S_A) \times D}_{\text{WFE(A) and SN(A)}} + \underbrace{(C_A + C_{AB} + C_B) \times D}_{\text{transfers}} \\ & + \underbrace{\frac{D}{N_{AB}} \times (S_A + S_B) \times D}_{\text{SN(A) and SN(B)}} + \underbrace{\frac{D}{N_B} \times (S_B + W_B) \times D}_{\text{SN(B) and WFE(B)}} \end{aligned}$$

In the considered infrastructure, the SN and WFE costs are the same at all sites, intra-site network capacities are equal, and intra-site transfers are free of cost. More precisely, $S_A = S_B$, and $W_A = W_B$, $N_A = N_B$, $C_A = C_B = 0$.

Given these concrete values, a double-indirect transfer will be more cost-effective than a direct one if:

$$
\begin{aligned}
C_{ind2}(A,B,A,B) &< C_{ind1}(A,B,x) \mid x \in \{A,B\} \\
2\frac{D}{N_A} \times (W_A + S_A) \times D + C_{AB} \times D + \frac{D}{N_{AB}} \times 2S_A \times D &< \frac{D}{N_{AB}} \times 2W_A \times D + C_{AB} \times D \\
\frac{2D^2 \times (W_A + S_A)}{N_A} + \frac{2D^2 \times S_A}{N_{AB}} &< \frac{2D^2 \times W_A}{N_{AB}} \\
\frac{W_A + S_A}{N_A} &< \frac{W_A - S_A}{N_{AB}} \\
N_{AB}(W_A + S_A) &< N_A(W_A - S_A) \\
S_A(N_A + N_{AB}) &< W_A(N_A - N_{AB}) \\
S_A &< W_A \frac{(N_A - N_{AB})}{(N_A + N_{AB})}
\end{aligned}
$$

In this particular constellation, the cost of storing data on a Storage node must thus be less than $\frac{8{,}500{,}000-500{,}000}{8{,}500{,}000+500{,}000} = \frac{8}{9}$ of the cost of storing data on a Workflow Engine for double-indirect transfers to become more cost-effective than direct transfers.

Evaluation

We have initially performed 50 planning runs with 500 generations each, where all of them started on an empty infrastructure. We employed a fitness function which assigned the same weights to execution speed (more precisely: optimizing for early termination) and execution cost. The WFE storage cost was set to be twice the cost of the Storage Nodes.

All of those runs achieved the optimal execution time. In other words, all of them found the correct partitioning strategy (assignment of WFEs to operation invocations), and maximized the requested resource usage. All of them chose an indirect data transfer strategy, but not all chose a double-indirect one. After further investigation, we repeated the evaluation, adjusting the cost factor to 25, 100, and 500 times, respectively. The results are shown in table Table 6.7.

Cost factor (WFE/SN)	min. cost ($)	max. cost ($)	avg. cost ($)	20th perc. ($)	80th perc. ($)	single indirect	double indirect
2	2.5222	2.5250	2.5243	2.5222	2.5250	72%	28%
25	2.5367	2.6272	2.5739	2.5368	2.6249	42%	58%
100	2.5840	2.9595	2.6429	2.5840	2.5841	16%	84%
500	2.8361	2.8364	2.8362	2.8361	2.8362	0%	100%

Table 6.7: Planning Scenario 2: Evaluation Results

Looking at the first line of the table, the results seem to be somewhat disappointing at first glance: only 14 of the 50 planning runs chose the strategy that we favored and

expected to be chosen. However, a look at the resulting costs provides an explanation: The best and worst results differ by only a quarter cent, or 0.1%. Essentially, the cost of such a workflow execution is completely dominated by the processing costs (in the order of cents per hour), and by network transmission costs (cents per GB transferred). Storage – cents/GB/month – contributes only a negligible amount to the overall price. In fact, for this constellation, $ 2.5099 are spent for CPU and network, and only around 1 – 2 cents for storage.

In subsequent evaluations, we successively raised the contribution of the cost of storing data at Workflow engines, thereby putting more pressure on the algorithm to prefer the "better" transfer strategy. As expected, the preference for double-indirect transfers gets higher as its gain in terms of cost increases. Thus, the results show that in principle, the planner is able to determine the best suited data transfer strategy by considering the cost of the workflow execution. However, the choice may only become very significant a) if temporary storage at WFEs is orders of magnitude more expensive than at SNs, which is not really realistic, or b) if data is stored for very long times during workflow execution.

6.1.4 Discussion

There are several reasons why we chose to use a Genetic Algorithm for the DWARFS planner component.

First, as previously mentioned, the combinatorial complexity of the problem space and its unpredictability (in the sense of the irregularity of existing reservations) effectively rule out exhaustive or analytical optimization approaches, and suggest the utilization of a meta-heuristic optimization approach.

Second, many previous publications have demonstrated that Genetic Algorithms are indeed an effective approach to solving scheduling problems at various levels of complexity, e.g., for scheduling jobs for multiprocessor systems and in the Grid [HAR94, GRH05, CDPEV05, PF05].

This assessment was also confirmed by our previous work [LS10], where the results reached an average effectiveness of 96.1% of a (known) optimum, and consistently produced good outcomes concerning both multi-objective optimizations (i.e., the user perspective) and system load/process partitioning (the system perspective).

Taking a critical stance on the evaluation results presented in this chapter, particularly in Section 6.1.2, they indicate that the Genetic Algorithm approach yields effective and enforceable plans, but falls short of *always* producing good (efficient) results. As discussed earlier, the reason is assumed to be a local optimum that the algorithm converges to, and which it is not easily leaving.

We do not currently know what exactly causes this problem, and how it could be avoided. One solution could be to adapt the chromosome mutation and/or crossover functions, by taking into account the "global" chromosome state and thus performing larger-scale adjustments. More radical approaches might include a redesign of the internal chromosome representation, or ultimately a complete switch to another class of meta-heuristic algorithms, e.g., to a Simulated Annealing [KGV83] or Tabu Search [Glo89] approach. However, we currently do not know how such changes would affect

the overall results, or the ability to avoid local minima. This topic is beyond the scope of this thesis and remains an open question for future work, for instance consisting of an evaluation of several meta-heuristics concerning their effectiveness and efficiency with regard to this planning scenario.

6.2 Enforcement Evaluation

While the previous section focused on the evaluation of the planner component, planning a workflow execution is only “half of the story” – in order for the DWARFS approach to be useful, one must also be able to actually execute the workflow according to the plan. In other words, the providers involved in the execution of a process must abide to the QoS guarantees that they committed to, and enforce the provision of the resources that have been reserved.

6.2.1 Scenario 1: Single Process on Amazon Web Services Infrastructure

In this scenario, we started with the simplest possible evaluation, namely to plan and execute a single workflow. We again used the Weather Forecast Workflow depicted in Figure 6.1, with one minor adaptation: all the operation execution times were divided by a factor of 5. This was only done to minimize the time required for the measurements, and does not affect the results in any other way. In terms of deployment, we also went for a minimalistic setup, with a single Workflow Engine on one node, and a single Operation Provider – providing all operations – on a second node. Both nodes were deployed at the same Amazon site (namely Ireland), and the Operation Provider was a dual-core machine (corresponding to a “medium” host type). The primary reason for employing such a minimal setup is that it is easier to saturate a small infrastructure (i.e.,, to force activities to actually execute in parallel, and thus to compete for resources where allocations have to be enforced). Actually deploying a large setup (such as the simulated one used for the planner evaluation) would have resulted in prohibitive costs, and since the enforcement is always performed in the same manner, we do not believe that it would have yielded a significantly different outcome.

Before presenting the results, let us shortly revisit a few important aspects concerning the enforcement and our setup. We mainly have to deal with the enforcement of transitory resources, namely network bandwidth and CPU. The enforcement of bandwidth limitations can only be performed on the sender side by throttling the amount of data that gets transmitted. The enforcement of CPU shares is done at the operation provider by monitoring CPU usages and adjusting priorities of the individual operations’ threads.

For the evaluations, we have implemented mocks of the individual operations. The data that is produced by each operation abides to the process definition in terms of data volume, but consists only of zeroes. Analogously, the CPU-intensive processing was simulated by repeatedly performing SHA-256 hash operations. We initially measured a the number of hash calculations that the node was capable to perform per minute.

Workflow Step	Planned	Actual	Deviation
Step 1: WFE transfer, Start Activity → Operation 1			
Transfer start	00:00:02	-00:11:40	-00:11:42
Transfer end	00:00:03	00:00:05	+00:00:02
Step 2: Invocation, Operation 1			
Upload start	00:01:06	00:00:05	-00:01:01
Upload end	00:01:09	00:01:06	-00:00:03
Execution start	00:01:09	00:01:06	-00:00:03
Execution end	00:01:29	00:01:21	-00:00:08
Download start	00:01:29	00:01:21	-00:00:08
Download end	00:02:57	00:02:56	-00:00:01
Step 3: WFE transfer, Start Activity → Operation 2			
Transfer start	00:00:01	-00:11:40	-00:11:41
Transfer end	00:00:02	00:00:06	+00:00:04
Step 4: Invocation, Operation 2			
Upload start	00:00:02	00:00:06	+00:00:04
Upload end	00:00:03	00:00:06	+00:00:03
Execution start	00:00:03	00:00:06	+00:00:03
Execution end	00:00:11	00:00:14	+00:00:03
Download start	00:00:11	00:00:14	+00:00:03
Download end	00:01:31	00:01:30	-00:00:01
Step 5: WFE transfer, Start Activity → Operation 3			
Transfer start	00:00:01	-00:11:41	-00:11:42
Transfer end	00:00:02	00:00:03	+00:00:01
Step 6: Invocation, Operation 3			
Upload start	00:00:20	00:00:03	-00:00:17
Upload end	00:00:23	00:00:20	-00:00:03
Execution start	00:00:23	00:00:20	-00:00:03
Execution end	00:00:27	00:00:27	-00:00:00
Download start	00:00:27	00:00:27	-00:00:00
Download end	00:01:29	00:01:28	-00:00:01
Step 7: WFE transfer, Start Activity → Operation 4			
Transfer start	00:00:01	-00:11:41	-00:11:42
Transfer end	00:00:02	00:00:05	+00:00:03
Step 8: Invocation, Operation 4			
Upload start	00:01:12	00:00:06	-00:01:06
Upload end	00:01:15	00:01:12	-00:00:03
Execution start	00:01:15	00:01:12	-00:00:03
Execution end	00:01:31	00:01:27	-00:00:04
Download start	00:01:31	00:01:27	-00:00:04
Download end	00:03:06	00:03:05	-00:00:01
Step 9: WFE transfer, Start Activity → Operation 7			
Transfer start	00:00:02	-00:11:40	-00:11:42
Transfer end	00:00:03	00:00:04	+00:00:01
Step 10: Invocation, Operation 7			
Upload start	00:02:27	00:00:04	-00:02:23
Upload end	00:02:28	00:02:27	-00:00:01
Execution start	00:02:28	00:02:27	-00:00:01
Execution end	00:02:57	00:02:40	-00:00:17
Download start	00:02:57	00:02:40	-00:00:17
Download end	00:04:12	00:04:11	-00:00:01
Step 11: WFE transfer, Operation 1 → Operation 5			
Transfer start	00:02:57	00:02:57	-00:00:00
Transfer end	00:11:42	00:11:41	-00:00:01
Step 12: Invocation, Operation 5			
Upload start	00:11:42	00:11:41	-00:00:01
Upload end	00:12:22	00:12:22	-00:00:00
Execution start	00:12:22	00:12:22	-00:00:00
Execution end	00:15:36	00:14:23	-00:01:13
Download start	00:15:36	00:14:23	-00:01:13
Download end	00:16:57	00:16:56	-00:00:01
Step 13: WFE transfer, Operation 1 → Operation 6			
Transfer start	00:02:57	00:02:57	-00:00:00
Transfer end	00:03:06	00:03:04	-00:00:02
Step 14: WFE transfer, Operation 2 → Operation 6			
Transfer start	00:01:31	00:01:32	+00:00:01
Transfer end	00:01:39	00:01:38	-00:00:01
Step 15: WFE transfer, Operation 3 → Operation 6			
Transfer start	00:01:29	00:01:29	-00:00:00
Transfer end	00:01:42	00:01:41	-00:00:01
Step 16: WFE transfer, Operation 4 → Operation 6			
Transfer start	00:03:06	00:03:06	-00:00:00
Transfer end	00:03:14	00:03:12	-00:00:02
Step 17: Invocation, Operation 6			
Upload start	00:03:14	00:03:12	-00:00:02
Upload end	00:06:05	00:06:05	-00:00:00
Execution start	00:06:05	00:06:05	-00:00:00
Execution end	00:18:05	00:18:12	+00:00:07
Download start	00:18:05	00:18:12	+00:00:07
Download end	00:22:34	00:22:43	+00:00:09
Step 18: WFE transfer, Operation 5 → Operation 8			
Transfer start	00:16:57	00:16:56	-00:00:01
Transfer end	00:17:16	00:17:14	-00:00:02
Step 19: WFE transfer, Operation 6 → Operation 8			
Transfer start	00:22:34	00:22:43	+00:00:09
Transfer end	00:23:10	00:23:09	-00:00:01
Step 20: WFE transfer, Operation 7 → Operation 8			
Transfer start	00:04:12	00:04:12	-00:00:00
Transfer end	00:04:22	00:04:21	-00:00:01
Step 21: Invocation, Operation 8			
Upload start	00:23:10	00:23:09	-00:00:01
Upload end	00:28:58	00:28:58	-00:00:00
Execution start	00:28:58	00:28:58	-00:00:00
Execution end	00:36:58	00:37:02	+00:00:04
Download start	00:36:58	00:37:02	+00:00:04
Download end	00:59:21	01:00:01	+00:00:40
Step 22: WFE transfer, Operation 8 → Operation 9			
Transfer start	00:59:21	01:00:01	+00:00:40
Transfer end	01:02:20	01:02:19	-00:00:01
Step 23: Invocation, Operation 9			
Upload start	01:02:20	01:02:19	-00:00:01
Upload end	01:24:43	01:25:32	+00:00:49
Execution start	01:24:43	01:25:32	+00:00:49
Execution end	01:40:43	01:41:41	+00:00:58
Download start	01:40:43	01:41:41	+00:00:58
Download end	02:03:06	02:03:43	+00:00:37
Step 24: WFE transfer, Operation 9 → Operation 10			
Transfer start	02:03:06	02:03:43	+00:00:37
Transfer end	02:06:05	02:06:04	-00:00:01
Step 25: Invocation, Operation 10			
Upload start	02:06:05	02:06:04	-00:00:01
Upload end	02:28:28	02:29:16	+00:00:48
Execution start	02:28:28	02:29:16	+00:00:48
Execution end	02:36:28	02:37:17	+00:00:49
Download start	02:36:28	02:37:17	+00:00:49
Download end	02:45:25	02:45:45	+00:00:20
Step 26: WFE transfer, Operation 10 → Operation 11			
Transfer start	02:45:25	02:45:45	+00:00:20
Transfer end	02:46:37	02:46:36	-00:00:01
Step 27: Invocation, Operation 11			
Upload start	02:46:37	02:46:36	-00:00:01
Upload end	02:55:34	02:55:53	+00:00:19
Execution start	02:55:34	02:55:53	+00:00:19
Execution end	02:55:46	02:56:05	+00:00:19
Download start	02:55:46	02:56:05	+00:00:19
Download end	02:55:50	02:56:06	+00:00:16
Step 28: WFE transfer, Operation 11 → End Activity			
Transfer start	02:55:50	02:56:06	+00:00:16
Transfer end	02:55:51	02:56:07	+00:00:16

Table 6.8: Enforcement Scenario 1: Evaluation Results

Operation invocations then would simply scale that number up according to the requested operation, and perform the resulting amount of hash calculations. Thus, while the workflow did not actually "do anything meaningful", this setup still simulated real behavior – following the characteristics of the weather forecast workflow – and allowed to evaluate the effectiveness of the enforcement.

The results for this scenario are shown in Table 6.8. For each workflow step, the planned and actual start and end timestamps are shown, as well as the deviation (the difference between the plan and the actual execution). Because this is the most important measure, the data in this column are presented in dark green if the actual execution was ahead of the planned time, and in dark orange if it was delayed behind the planned time.

- The "planned" and "actual" timestamps are given in HH:MM:SS format, and are relative to the actually planned start of the workflow.
- All "actual" values have been rounded to the nearest full second.
- The negative values found at the beginning of the process are caused by the client computer having started the execution too early.
- Whenever the actual execution was ahead of the plan, the enforcement components ensured that the execution was delayed until the planned time was reached.

Discussion

As can be seen from the results, at first glance, the enforcement generally works reasonably well – the total duration of the entire process was just under 3 hours, and it finished with a delay of 16 seconds. On the other hand, in a real-world setting, even a delay of a single second could result in major problems, because providers have only committed to provide resources exactly for the planned timeslots, and might refuse to allocate resources beyond the reserved time. It is therefore necessary to analyze where such delays occur.

Taking a closer look at the results, one can distinguish three different cases where delays occur. The first case occurs at the very beginning of the process execution, namely in the first 15 seconds. We believe these to be related to the workflow engine performing some initial setup (e.g., fetching the WSDL documents for the invocations). These delays do not exceed 4 seconds, and could be eliminated by adjusting the implementation of the workflow engine.

The second case occurs for instance during the invocation of Operation 6 (Workflow step 17), where the actual execution starts on time, but finishes 7 seconds late. In this case, the CPU took longer than expected to perform the calculations. In other words, the provider had "overestimated" the capacity of the CPU, and would have to adjust the co-allocation constraints.

The third case is the most serious, and occurs most prominently starting with the download of data from Operation 8 through the end of the workflow: Every data download (from the Operation Provider to the Workflow Engine), as well as every data upload (from the WFE to the OP) finishes late. We first believed this to be related to the

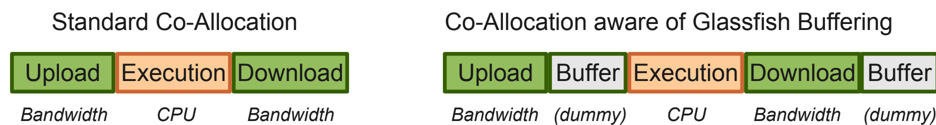


Figure 6.9: Enforcement: Accounting for Buffering Behavior

Amazon infrastructure, which does not give any explicit guarantees concerning network bandwidth, and could simply have been overloaded. Therefore, we repeated the evaluation in a more controlled, local network, but the issues persisted. In fact, these issues are only happening in the parts of the workflow where large data (1 GB or more) are shipped, and the delays are roughly proportional to the data volume.

The delays are caused by the Application Server (Glassfish) buffering large data to disk. Even if the data sender is perfectly abiding to the plan and transmitting data exactly at the reserved rate, additional buffering occurs only after data has been completely transmitted. While this explains the delays, we did not find a way to fix that behavior. In order to still be able to perform meaningful evaluations, we therefore had to account for this particular behavior of the application server. The solution that we adopted is entirely consistent with the model, and is shown in Figure 6.9: instead of an operation invocation only entailing upload, execution, and download stages, it contains an additional "buffer" stage after each data transfer. This is achieved by modifying the co-allocation constraints for every operation invocation, by adding additional allocation constraints for unique (dummy) resources.

6.2.2 Scenario 2: Multiple Processes on Local Infrastructure

After applying the abovementioned adaptations to account for deficiencies of the tools used in the implementation, the enforcement did not exhibit delays related to buffering anymore. Thus, we proceeded to the evaluation of the enforcement of multiple simultanous processes. This evaluation was done on a local infrastructure, with the process definition shown in Figure 6.1 (with unmodified operation durations). We again employed a single "medium" Operation Provider, and a single Workflow Engine.

We consecutively scheduled three instances of the workflow, for all of which early termination was used as the planning goal, and all of which were planned to start at an identical timestamp (if possible). Naturally, the first process had the shortest total duration, as it was planning on an infrastructure with no initially present reservations, while the subsequently planned processes had to account for reservations made earlier. However, the execution of those three processes was still overlapping, thus requiring correct resource enforcement.

The results are depicted in Tables 6.9, 6.10, and 6.11. With the single exception of a delay of 2 seconds when executing Operation 1 of Process 1 (which we cannot explain), the actual executions are perfectly abiding to the plans. There are still delays present during large data transfers, but these are now expected and absorbed by the additional "buffer" allocations. For simplicity's sake, we used 180 seconds as the duration for all those buffers (because that number was sufficient for all delays observed in this scenario), but they could also be adjusted to take into account the amount of data.

Workflow Step	Planned	Actual	Deviation
Step 1: WFE transfer, Start Activity → Operation 1			
Transfer start	00:00:01	-00:23:55	-00:23:56
Transfer end	00:00:02	00:00:01	-00:00:01
Step 2: Invocation, Operation 1			
Upload start	00:00:02	00:00:02	-00:00:00
Upload end	00:00:03	00:00:02	-00:00:01
Execution start	00:03:03	00:00:02	-00:03:01
Execution end	00:04:03	00:04:05	+00:00:02
Download start	00:04:03	00:04:05	+00:00:02
Download end	00:09:39	00:09:37	-00:00:02
Step 3: WFE transfer, Start Activity → Operation 2			
Transfer start	00:00:02	-00:23:55	-00:23:57
Transfer end	00:00:03	00:00:02	-00:00:01
Step 4: Invocation, Operation 2			
Upload start	00:00:03	00:00:02	-00:00:01
Upload end	00:00:04	00:00:03	-00:00:01
Execution start	00:03:04	00:00:03	-00:03:01
Execution end	00:03:53	00:03:46	-00:00:07
Download start	00:03:53	00:03:46	-00:00:07
Download end	00:11:57	00:11:56	-00:00:01
Step 5: WFE transfer, Start Activity → Operation 3			
Transfer start	00:00:02	-00:23:55	-00:23:57
Transfer end	00:00:03	00:00:02	-00:00:01
Step 6: Invocation, Operation 3			
Upload start	00:05:53	00:00:02	-00:05:51
Upload end	00:05:54	00:05:53	-00:00:01
Execution start	00:08:54	00:05:53	-00:03:01
Execution end	00:09:39	00:09:14	-00:00:25
Download start	00:09:39	00:09:14	-00:00:25
Download end	00:17:08	00:17:06	-00:00:02
Step 7: WFE transfer, Start Activity → Operation 4			
Transfer start	00:00:03	-00:23:55	-00:23:58
Transfer end	00:00:04	00:00:03	-00:00:01
Step 8: Invocation, Operation 4			
Upload start	00:00:59	00:00:03	-00:00:56
Upload end	00:01:00	00:00:59	-00:00:01
Execution start	00:04:00	00:00:59	-00:03:01
Execution end	00:05:29	00:05:02	-00:00:27
Download start	00:05:29	00:05:02	-00:00:27
Download end	00:17:49	00:17:48	-00:00:01
Step 9: WFE transfer, Start Activity → Operation 7			
Transfer start	00:00:04	-00:23:56	-00:24:00
Transfer end	00:00:05	00:00:04	-00:00:01
Step 10: Invocation, Operation 7			
Upload start	00:08:33	00:00:04	-00:08:29
Upload end	00:08:34	00:08:33	-00:00:01
Execution start	00:11:34	00:08:33	-00:03:01
Execution end	00:17:08	00:12:33	-00:04:35
Download start	00:17:08	00:12:33	-00:04:35
Download end	00:25:15	00:25:14	-00:00:01
Step 11: WFE transfer, Operation 1 → Operation 5			
Transfer start	00:12:39	00:09:38	-00:03:01
Transfer end	00:14:24	00:14:23	-00:00:01
Step 12: Invocation, Operation 5			
Upload start	00:35:01	00:14:23	-00:20:38
Upload end	00:43:22	00:43:21	-00:00:01
Execution start	00:46:22	00:43:21	-00:03:01
Execution end	01:02:30	00:56:36	-00:05:54
Download start	01:02:30	00:56:36	-00:05:54
Download end	01:10:13	01:10:12	-00:00:01
Step 13: WFE transfer, Operation 1 → Operation 6			
Transfer start	00:12:39	00:09:38	-00:03:01
Transfer end	00:13:32	00:13:31	-00:00:01
Step 14: WFE transfer, Operation 2 → Operation 6			
Transfer start	00:14:57	00:11:56	-00:03:01
Transfer end	00:15:47	00:15:46	-00:00:01
Step 15: WFE transfer, Operation 3 → Operation 6			
Transfer start	00:20:08	00:17:07	-00:03:01
Transfer end	00:20:50	00:20:49	-00:00:01
Step 16: WFE transfer, Operation 4 → Operation 6			
Transfer start	00:20:50	00:17:48	-00:03:02
Transfer end	00:21:32	00:21:31	-00:00:01
Step 17: Invocation, Operation 6			
Upload start	00:21:32	00:21:31	-00:00:01
Upload end	00:34:54	00:34:56	+00:00:02
Execution start	00:37:54	00:34:56	-00:02:58
Execution end	01:37:54	01:37:36	-00:00:18
Download start	01:37:54	01:37:36	-00:00:18
Download end	01:58:58	01:59:01	+00:00:03
Step 18: WFE transfer, Operation 5 → Operation 8			
Transfer start	01:13:13	01:10:12	-00:03:01
Transfer end	01:14:30	01:14:29	-00:00:01
Step 19: WFE transfer, Operation 6 → Operation 8			
Transfer start	02:01:58	01:59:01	-00:02:57
Transfer end	02:05:33	02:05:32	-00:00:01
Step 20: WFE transfer, Operation 7 → Operation 8			
Transfer start	00:28:15	00:25:14	-00:03:01
Transfer end	00:29:15	00:29:14	-00:00:01
Step 21: Invocation, Operation 8			
Upload start	02:05:33	02:05:32	-00:00:01
Upload end	02:32:47	02:33:11	+00:00:24
Execution start	02:35:47	02:33:11	-00:02:36
Execution end	03:15:47	03:15:01	-00:00:46
Download start	03:15:47	03:15:01	-00:00:46
Download end	05:01:04	05:02:53	+00:01:49
Step 22: WFE transfer, Operation 8 → Operation 9			
Transfer start	05:04:04	05:02:54	-00:01:10
Transfer end	05:21:58	05:21:57	-00:00:01
Step 23: Invocation, Operation 9			
Upload start	05:21:58	05:21:57	-00:00:01
Upload end	07:07:15	07:07:49	+00:00:34
Execution start	07:10:15	07:07:49	-00:02:26
Execution end	08:30:15	08:29:14	-00:01:01
Download start	08:30:15	08:29:14	-00:01:01
Download end	10:15:32	10:17:23	+00:01:51
Step 24: WFE transfer, Operation 9 → Operation 10			
Transfer start	10:18:32	10:17:24	-00:01:08
Transfer end	10:36:26	10:36:25	-00:00:01
Step 25: Invocation, Operation 10			
Upload start	10:36:26	10:36:25	-00:00:01
Upload end	12:21:43	12:22:20	+00:00:37
Execution start	12:24:43	12:22:20	-00:02:23
Execution end	13:04:43	13:04:13	-00:00:30
Download start	13:04:43	13:04:13	-00:00:30
Download end	13:46:50	13:47:33	+00:00:43
Step 26: WFE transfer, Operation 10 → Operation 11			
Transfer start	13:49:50	13:47:33	-00:02:17
Transfer end	13:57:00	13:56:59	-00:00:01
Step 27: Invocation, Operation 11			
Upload start	13:57:00	13:56:59	-00:00:01
Upload end	14:39:07	14:39:22	+00:00:15
Execution start	14:42:07	14:39:22	-00:02:45
Execution end	14:43:07	14:43:06	-00:00:01
Download start	14:43:07	14:43:06	-00:00:01
Download end	14:43:20	14:43:19	-00:00:01
Step 28: WFE transfer, Operation 11 → End Activity			
Transfer start	14:46:20	14:43:19	-00:03:01
Transfer end	14:46:26	14:46:24	-00:00:02

Table 6.9: Enforcement Scenario 2, Process 1: Evaluation Results

Workflow Step	Planned	Actual	Deviation
Step 1: WFE transfer, Start Activity → Operation 1			
Transfer start	00:00:03	-00:23:44	-00:23:47
Transfer end	00:00:04	00:00:03	-00:00:01
Step 2: Invocation, Operation 1			
Upload start	00:34:54	00:00:03	-00:34:51
Upload end	00:34:57	00:34:55	-00:00:02
Execution start	00:37:57	00:34:55	-00:03:02
Execution end	00:40:05	00:39:00	-00:01:05
Download start	00:40:05	00:39:00	-00:01:05
Download end	00:50:02	00:50:01	-00:00:01
Step 3: WFE transfer, Start Activity → Operation 2			
Transfer start	00:00:06	-00:23:44	-00:23:50
Transfer end	00:00:07	00:00:06	-00:00:01
Step 4: Invocation, Operation 2			
Upload start	00:12:19	00:00:06	-00:12:13
Upload end	00:12:20	00:12:19	-00:00:01
Execution start	00:15:20	00:12:19	-00:03:01
Execution end	00:17:49	00:15:59	-00:01:50
Download start	00:17:49	00:15:59	-00:01:50
Download end	00:22:36	00:22:35	-00:00:01
Step 5: WFE transfer, Start Activity → Operation 3			
Transfer start	00:00:05	-00:23:44	-00:23:49
Transfer end	00:00:06	00:00:05	-00:00:01
Step 6: Invocation, Operation 3			
Upload start	00:18:28	00:00:05	-00:18:23
Upload end	00:18:29	00:18:28	-00:00:01
Execution start	00:21:29	00:18:28	-00:03:01
Execution end	00:22:36	00:21:49	-00:00:47
Download start	00:22:36	00:21:49	-00:00:47
Download end	00:30:05	00:30:04	-00:00:01
Step 7: WFE transfer, Start Activity → Operation 4			
Transfer start	00:00:04	-00:23:44	-00:23:48
Transfer end	00:00:05	00:00:04	-00:00:01
Step 8: Invocation, Operation 4			
Upload start	00:20:20	00:00:04	-00:20:16
Upload end	00:20:21	00:20:20	-00:00:01
Execution start	00:23:21	00:20:20	-00:03:01
Execution end	00:25:15	00:24:21	-00:00:54
Download start	00:25:15	00:24:21	-00:00:54
Download end	00:35:03	00:35:02	-00:00:01
Step 9: WFE transfer, Start Activity → Operation 7			
Transfer start	00:00:05	-00:23:44	-00:23:49
Transfer end	00:00:06	00:00:05	-00:00:01
Step 10: Invocation, Operation 7			
Upload start	00:59:29	00:00:05	-00:59:24
Upload end	00:59:30	00:59:29	-00:00:01
Execution start	01:02:30	00:59:29	-00:03:01
Execution end	02:42:30	01:12:51	-01:29:39
Download start	02:42:30	01:12:51	-01:29:39
Download end	02:47:15	02:47:14	-00:00:01
Step 11: WFE transfer, Operation 1 → Operation 5			
Transfer start	00:53:05	00:50:01	-00:03:04
Transfer end	00:53:44	00:53:43	-00:00:01
Step 12: Invocation, Operation 5			
Upload start	00:53:44	00:53:43	-00:00:01
Upload end	00:58:48	00:58:47	-00:00:01
Execution start	01:01:48	00:58:47	-00:03:01
Execution end	01:23:40	01:11:51	-00:11:49
Download start	01:23:40	01:11:51	-00:11:49
Download end	01:30:41	01:30:40	-00:00:01
Step 13: WFE transfer, Operation 1 → Operation 6			
Transfer start	00:53:02	00:50:01	-00:03:01
Transfer end	00:54:29	00:54:28	-00:00:01
Step 14: WFE transfer, Operation 2 → Operation 6			
Transfer start	00:25:36	00:22:35	-00:03:01
Transfer end	00:26:54	00:26:53	-00:00:01
Step 15: WFE transfer, Operation 3 → Operation 6			
Transfer start	00:33:05	00:30:04	-00:03:01
Transfer end	00:34:56	00:34:54	-00:00:02
Step 16: WFE transfer, Operation 4 → Operation 6			
Transfer start	00:38:03	00:35:02	-00:03:01
Transfer end	00:39:12	00:39:11	-00:00:01
Step 17: Invocation, Operation 6			
Upload start	01:07:14	00:54:28	-00:12:46
Upload end	01:25:18	01:25:18	-00:00:00
Execution start	01:28:18	01:25:18	-00:03:00
Execution end	02:47:15	02:27:47	-00:19:28
Download start	02:47:15	02:27:47	-00:19:28
Download end	03:09:39	03:09:40	+00:00:01
Step 18: WFE transfer, Operation 5 → Operation 8			
Transfer start	01:33:41	01:30:40	-00:03:01
Transfer end	01:35:38	01:35:37	-00:00:01
Step 19: WFE transfer, Operation 6 → Operation 8			
Transfer start	03:12:39	03:09:40	-00:02:59
Transfer end	03:25:55	03:25:53	-00:00:02
Step 20: WFE transfer, Operation 7 → Operation 8			
Transfer start	02:50:15	02:47:14	-00:03:01
Transfer end	02:51:32	02:51:31	-00:00:01
Step 21: Invocation, Operation 8			
Upload start	03:49:42	03:25:53	-00:23:49
Upload end	04:18:04	04:18:12	+00:00:08
Execution start	04:21:04	04:18:12	-00:02:52
Execution end	05:01:04	05:00:43	-00:00:21
Download start	05:01:04	05:00:43	-00:00:21
Download end	07:48:10	07:50:13	+00:02:03
Step 22: WFE transfer, Operation 8 → Operation 9			
Transfer start	07:51:10	07:50:13	-00:00:57
Transfer end	08:09:04	08:09:03	-00:00:01
Step 23: Invocation, Operation 9			
Upload start	08:09:04	08:09:03	-00:00:01
Upload end	09:54:21	09:54:56	+00:00:35
Execution start	09:57:21	09:54:56	-00:02:25
Execution end	11:17:21	11:16:25	-00:00:56
Download start	11:17:21	11:16:25	-00:00:56
Download end	13:02:38	13:04:27	+00:01:49
Step 24: WFE transfer, Operation 9 → Operation 10			
Transfer start	13:05:38	13:04:27	-00:01:11
Transfer end	13:36:30	13:36:28	-00:00:02
Step 25: Invocation, Operation 10			
Upload start	14:39:07	13:36:28	-01:02:39
Upload end	16:24:24	16:25:01	+00:00:37
Execution start	16:27:24	16:25:01	-00:02:23
Execution end	17:07:24	17:06:45	-00:00:39
Download start	17:07:24	17:06:45	-00:00:39
Download end	17:49:31	17:50:03	+00:00:32
Step 26: WFE transfer, Operation 10 → Operation 11			
Transfer start	17:52:31	17:50:03	-00:02:28
Transfer end	17:59:41	17:59:40	-00:00:01
Step 27: Invocation, Operation 11			
Upload start	17:59:41	17:59:40	-00:00:01
Upload end	18:41:48	18:42:03	+00:00:15
Execution start	18:44:48	18:42:03	-00:02:45
Execution end	18:45:48	18:45:47	-00:00:01
Download start	18:45:48	18:45:47	-00:00:01
Download end	18:46:01	18:45:59	-00:00:02
Step 28: WFE transfer, Operation 11 → End Activity			
Transfer start	18:49:01	18:45:59	-00:03:02
Transfer end	18:49:04	18:49:02	-00:00:02

Table 6.10: Enforcement Scenario 2, Process 2: Evaluation Results

Workflow Step	Planned	Actual	Deviation
Step 1: WFE transfer, Start Activity → Operation 1			
Transfer start	01:44:07	-00:23:20	-02:07:27
Transfer end	01:44:08	01:44:07	-00:00:01
Step 2: Invocation, Operation 1			
Upload start	01:54:41	01:44:07	-00:10:34
Upload end	01:54:42	01:54:41	-00:00:01
Execution start	01:57:42	01:54:41	-00:03:01
Execution end	01:58:58	01:58:42	-00:00:16
Download start	01:58:58	01:58:42	-00:00:16
Download end	02:05:01	02:05:00	-00:00:01
Step 3: WFE transfer, Start Activity → Operation 2			
Transfer start	01:44:08	-00:23:20	-02:07:28
Transfer end	01:44:09	01:44:08	-00:00:01
Step 4: Invocation, Operation 2			
Upload start	02:01:23	01:44:08	-00:17:15
Upload end	02:01:24	02:01:23	-00:00:01
Execution start	02:04:24	02:01:23	-00:03:01
Execution end	02:07:26	02:05:04	-00:02:22
Download start	02:07:26	02:05:04	-00:02:22
Download end	02:12:35	02:12:34	-00:00:01
Step 5: WFE transfer, Start Activity → Operation 3			
Transfer start	01:44:08	-00:23:20	-02:07:28
Transfer end	01:44:09	01:44:08	-00:00:01
Step 6: Invocation, Operation 3			
Upload start	01:55:57	01:44:08	-00:11:49
Upload end	01:55:58	01:55:57	-00:00:01
Execution start	01:58:58	01:55:57	-00:03:01
Execution end	01:59:28	01:59:18	-00:00:10
Download start	01:59:28	01:59:18	-00:00:10
Download end	02:07:26	02:07:24	-00:00:02
Step 7: WFE transfer, Start Activity → Operation 4			
Transfer start	01:44:09	-00:23:20	-02:07:29
Transfer end	01:44:10	01:44:09	-00:00:01
Step 8: Invocation, Operation 4			
Upload start	01:59:33	01:44:09	-00:15:24
Upload end	01:59:34	01:59:33	-00:00:01
Execution start	02:02:34	01:59:33	-00:03:01
Execution end	02:05:01	02:03:33	-00:01:28
Download start	02:05:01	02:03:33	-00:01:28
Download end	02:15:34	02:15:33	-00:00:01
Step 9: WFE transfer, Start Activity → Operation 7			
Transfer start	01:44:09	-00:23:20	-02:07:29
Transfer end	01:44:10	01:44:09	-00:00:01
Step 10: Invocation, Operation 7			
Upload start	07:43:45	01:44:09	-05:59:36
Upload end	07:43:58	07:43:56	-00:00:02
Execution start	07:46:58	07:43:56	-00:03:02
Execution end	07:48:10	07:47:59	-00:00:11
Download start	07:48:10	07:47:59	-00:00:11
Download end	07:53:53	07:53:54	+00:00:01
Step 11: WFE transfer, Operation 1 → Operation 5			
Transfer start	02:08:01	02:05:00	-00:03:01
Transfer end	02:09:43	02:09:42	-00:00:01
Step 12: Invocation, Operation 5			
Upload start	02:46:04	02:09:42	-00:36:22
Upload end	02:54:53	02:54:53	-00:00:00
Execution start	02:57:53	02:54:53	-00:03:00
Execution end	03:09:39	03:08:02	-00:01:37
Download start	03:09:39	03:08:02	-00:01:37
Download end	03:15:10	03:15:09	-00:00:01
Step 13: WFE transfer, Operation 1 → Operation 6			
Transfer start	02:08:01	02:05:00	-00:03:01
Transfer end	02:10:24	02:10:23	-00:00:01
Step 14: WFE transfer, Operation 2 → Operation 6			
Transfer start	02:15:35	02:12:34	-00:03:01
Transfer end	02:16:20	02:16:19	-00:00:01
Step 15: WFE transfer, Operation 3 → Operation 6			
Transfer start	02:10:26	02:07:25	-00:03:01
Transfer end	02:12:20	02:12:18	-00:00:02
Step 16: WFE transfer, Operation 4 → Operation 6			
Transfer start	02:18:34	02:15:33	-00:03:01
Transfer end	02:20:06	02:20:04	-00:00:02
Step 17: Invocation, Operation 6			
Upload start	03:29:41	02:20:04	-01:09:37
Upload end	03:44:53	03:44:56	+00:00:03
Execution start	03:47:53	03:44:56	-00:02:57
Execution end	05:01:04	04:47:20	-00:13:44
Download start	05:01:04	04:47:20	-00:13:44
Download end	06:08:59	06:09:29	+00:00:30
Step 18: WFE transfer, Operation 5 → Operation 8			
Transfer start	03:18:10	03:15:09	-00:03:01
Transfer end	03:24:44	03:24:43	-00:00:01
Step 19: WFE transfer, Operation 6 → Operation 8			
Transfer start	06:11:59	06:09:29	-00:02:30
Transfer end	06:15:34	06:15:33	-00:00:01
Step 20: WFE transfer, Operation 7 → Operation 8			
Transfer start	08:09:04	07:53:54	-00:15:10
Transfer end	08:11:10	08:11:09	-00:00:01
Step 21: Invocation, Operation 8			
Upload start	12:25:35	08:11:09	-04:14:26
Upload end	13:28:54	13:29:02	+00:00:08
Execution start	13:31:54	13:29:02	-00:02:52
Execution end	14:43:20	14:11:19	-00:32:01
Download start	14:43:20	14:11:19	-00:32:01
Download end	16:54:56	16:56:45	+00:01:49
Step 22: WFE transfer, Operation 8 → Operation 9			
Transfer start	16:57:56	16:56:46	-00:01:10
Transfer end	17:29:54	17:29:53	-00:00:01
Step 23: Invocation, Operation 9			
Upload start	18:41:48	17:29:53	-01:11:55
Upload end	20:27:05	20:27:42	+00:00:37
Execution start	20:30:05	20:27:42	-00:02:23
Execution end	21:50:05	21:48:53	-00:01:12
Download start	21:50:05	21:48:53	-00:01:12
Download end	23:35:22	23:37:15	+00:01:53
Step 24: WFE transfer, Operation 9 → Operation 10			
Transfer start	23:38:22	23:37:15	-00:01:07
Transfer end	23:56:16	23:56:15	-00:00:01
Step 25: Invocation, Operation 10			
Upload start	23:56:16	23:56:15	-00:00:01
Upload end	25:41:33	25:42:11	+00:00:38
Execution start	25:44:33	25:42:11	-00:02:22
Execution end	26:24:33	26:23:54	-00:00:39
Download start	26:24:33	26:23:54	-00:00:39
Download end	27:06:40	27:07:10	+00:00:30
Step 26: WFE transfer, Operation 10 → Operation 11			
Transfer start	27:09:40	27:07:10	-00:02:30
Transfer end	27:16:50	27:16:49	-00:00:01
Step 27: Invocation, Operation 11			
Upload start	27:16:50	27:16:49	-00:00:01
Upload end	27:58:57	27:59:12	+00:00:15
Execution start	28:01:57	27:59:12	-00:02:45
Execution end	28:02:57	28:02:56	-00:00:01
Download start	28:02:57	28:02:56	-00:00:01
Download end	28:03:10	28:03:09	-00:00:01
Step 28: WFE transfer, Operation 11 → End Activity			
Transfer start	28:06:10	28:03:09	-00:03:01
Transfer end	28:06:13	28:06:12	-00:00:01

Table 6.11: Enforcement Scenario 2, Process 3: Evaluation Results

6.2.3 Discussion

After applying the abovementioned workaround (which is only needed to account for behavior specific to the employed software stack), the resource enforcement turned out to be very good to excellent, and all processes were able to (almost) exactly abide to their schedules. However, minimal deviations (in the order of a few seconds) could still occur simply because of the dynamicity of the system (such as background processes not controllable by DWARFS), so we suggest to employ a slightly more conservative strategy which allows to absorb small delays. This could for instance be achieved by marginally underestimating the capacities of transitory resources, such as CPU or bandwidth (thus minimally overestimating the time required during execution).

7

Related Work

The purpose of the DWARFS system, as a whole, is the distributed execution of Scientific Workflows in a predictable manner by using Advance Reservations. In the previous chapters, we have described the details and concepts and methods that we employ to realize such a system. In this chapter, we will take a step back from the DWARFS system as such. Instead, we will look at the important aspects of the problem setting as such, and present and put into perspective other representative systems and approaches which tackle the same or similar, related, problems.

We start with a general overview of entire systems for workflow execution, while highlighting unique features, and similarities with (and differences to) DWARFS. Afterwards, we consider the problem of planning or scheduling workflow executions. Finally, we take a look at one particular aspect of execution – which in turn influences the planning –, namely the supervision of executing operations to predict their future behavior.

While the concepts involved are very similar, different research groups often use slightly different vocabulary to designate them. In the following, wherever possible and meaningful, we will employ the terminology used throughout this document.

7.1 Distributed and Scientific Workflow Systems

The single smallest common denominator for all workflow systems is that they are able to execute (orchestrate) workflows composed of individual activities, linked to each other via data flow or control flow dependencies. However, commonalities already end here, as there are many degrees of freedom concerning specific features. We shortly present the most important of these below.

Workflow Characteristics

Typically, a distinction is made between Business Workflows and Scientific Workflows. Apart from qualitative distinction by the different domain use cases (e.g., booking a flight, or performing an evaluation of experimental data), these workflow types also

differ “quantitatively”: while Business workflows are generally short-running (seconds, or minutes) and do not operate on large volumes of data, Scientific Workflows typically involve large datasets (e.g., Gigabytes and upwards) and consist of computationally expensive and thus long-running activities (in the order of hours to weeks). Put simply, Scientific Workflow Systems are aware of potentially very high resource requirements, and are prepared to handle them.

Provisioning Paradigm

In general, there are two very different possibilities to make use of (remotely) available processing capabilities. The first option is to send the executable program to the remote site, and to execute it there. This approach is used in traditional Grid Computing systems (where individual activities are usually termed jobs), and we will designate this type of usage by the term *Infrastructure as a Service* (IaaS). On the other end of the scale, there is what we will call *Software as a Service* (SaaS), where one merely invokes the functionality provided by the remote end. In this case, the caller sends the data to be processed to the provider, but the actual implementation of the functionality remains known to the provider only, and is essentially a black box to the user. This method is predominantly employed for the execution of Business Processes in purely Service Oriented Architectures.

There is a third possibility, which is actually used by existing Workflow Systems, and which could be described as consolidating, or fitting “in between”, the abovementioned approaches. To stay with established terminology, we will use the term *Platform as a Service* (PaaS) to refer to such systems. What we mean here is that the workflow system itself actually provides a “programming platform” to define the implementation of activities involved in a workflow. In principle, both IaaS- and SaaS- style invocations can thus be used, for instance by wrapping a Grid job submission or a SOAP call inside an activity.

Workflow Structure

There are two fundamental ways to model workflows (or more specifically: workflow definitions). One widely employed possibility is to represent a workflow as a Directed Acyclic Graph (DAG), thus disallowing loops in the control flow. The other alternative is to allow such loops. While this allows for more complex workflow structures, such (potentially unbounded) loops also pose significant problems for the execution planning.

Orchestration Locality

In the simplest case, the execution of a workflow instance is orchestrated by a single, centralized, Workflow Engine, which dispatches the individual activities to the respective operation providers and monitors and controls the progress of the entire workflow. In an SaaS setting, that usually consists of (multiple) remote invocations of Web Services. In an IaaS setting, it consists of (multiple) job submissions to remote sites. Finally, in a PaaS setting, the activities are usually executed inside the WFE process (or as child

processes on the same machine); however, as described above, the activities themselves may consist of remote invocations.

A decentralized orchestration uses multiple, cooperating Workflow Engines, which are responsible for handling individual fragments of the workflow. Information about the state of the execution (both in terms of control flow, and data) must thus be communicated between the participating WFEs.

Binding Flexibility

If there are multiple choices for executing a particular activity (i.e., multiple Operation Providers offering the same Operation, or multiple Grid nodes available for executing a Grid job), then one has to choose which of these to use. This selection – called binding – can generally be performed at different times. The earliest, and least flexible, possibility is at the time of defining the workflow, by simply hard-coding the instance to use. The latest possible time is at runtime, just in time before the actual invocation. Finally, the third option is to perform the bindings individually for each workflow instance, for example during a separate planning phase (as DWARFS does).

The abovementioned characteristics are the ones that we deem most relevant for the following observations. In [YB05], Yu and Buyya present a taxonomy of Workflow Systems for Grid Computing, which presents a detailed classification of various systems according to these and other characteristics.

According to the abovementioned criteria, the DWARFS system is geared at the execution of Scientific Workflows in an (exclusively) Service Oriented Infrastructure, thus using the SaaS provisioning paradigm. Workflows are represented as DAGs, and are orchestrated in a decentralized fashion by multiple WFEs. Finally, binding of activities (and assignment of WFEs) is performed during the planning phase prior to the execution – in fact, it corresponds to the establishment of the Advance Reservations.

In the world of Business Processes, to the best of our knowlegde, all major available products use a centralized coordination scheme. In [Kha08], Khalaf presents a method and system to support a decentralized execution of Business Processes. To that end, BPEL-D (BPEL with Data-links) is introduced as a variant of the Business Process Execution Language, allowing to partition existing workflow definitions into multiple fragments. The partitioning is manually provided by the user; the resulting fragments can then run on multiple cooperating (BPEL-D) WFEs, while maintaining the operational semantics of the original workflow. Because it is based on BPEL, the system supports non-DAG workflow structures and is geared at a SaaS provisioning. Activity binding in BPEL workflows is generally static (defined at process definition time), but allows for limited flexibility by (programmatically) re-assigning instance endpoints during the workflow execution.

Osiris [STS^{+}06] is a fully decentralized Process Management System. Workflows are defined as a Directed Acyclic Graph. Osiris is designed as a SOA and thus falls into the SaaS provisioning category. One unique feature of the system is that every participating node in fact also is a WFE, so that orchestration and activity invocation logically collapse into one: the execution is orchestrated by the nodes currently performing a workflow

activity. Osiris uses late binding, determining the most suitable node to continue the execution just in time from deployment and load information available through system-wide repositories. As the entire workflow execution state is shipped when transitioning from one node to another, and data size (and network locality) are not considered during the decision making, the system is not very well suited for workflows involving massive amounts of data.

JOpera [PA04] is a tool for designing and executing workflows. It is mostly targeting the SaaS paradigm (SOAP and REST Web Services are supported as first class citizens), but also includes PaaS aspects which allow to include local components (such as user-defined Java code, or job submissions to Grid environments). The workflow structure is not limited to DAGs. JOpera supports many binding possibilities, including the unique possibility to flexibly determine bindings by reflection, i.e., by introspecting the current process state [PA05]. The system uses centralized orchestration; even though it supports scaling out (for instance by deploying multiple WFE instances on a cluster), it does not seem to support a truely decentralized coordination of process orchestration.

Taverna [OAF+04] was designed as a "tool for the composition and enactment of bioinformatics workflows". It can best be described as using the PaaS paradigm: workflow orchestration is handled by a single, local, engine. Activities are thus – in principle – run locally, but may make use of so-called "processors", which can interface with SOAP Web Services (and with traditional Grid sites [KMB08]). Taverna supports workflows defined as DAGs, and uses late (just in time) binding, where available instances can be looked up in UDDI registries or ontology directories.

Kepler [ABJ+04] is another system that falls into the PaaS category, in that it provides a platform where activities can be defined and "plugged together". The system was initially designed to target classic Grid environments, and has then been expanded to also support SOAs (whereas Taverna evolved in the opposite direction). Kepler supports non-DAG workflows, and uses a centralized execution environment. Kepler employs static (definition-time) binding, but supports re-assignment to alternative instances in the case of failures.

Askalon [FJP+05] is mainly targeted at grid workflow applications and thus uses the IaaS paradigm. It employs decentralized orchestration and supports non-DAG workflows. In contrast to the abovementioned systems, it can be aware of the resource requirements of workflows, and provides planning and advance reservation capabilities through a system termed GridARM [SF05]. The main difference to the DWARFS approach lies in the different provisioning paradigm: Askalon focuses on IaaS – thus, the user is responsible for predicting the execution characteristics. DWARFS instead focuses on SaaS, so that this responsibility lies with the operation providers.

7.2 Planning and Advance Reservations

In this section, we will cover a number of interesting approaches related to various aspects of execution planning. While none of them considers the exact same scope as our work, there is a certain overlap in the involved concepts. Note that unless stated otherwise, all of these approaches target the IaaS provisioning paradigm – in other words,

they assume that in principle, any activity (i.e., grid job) can be assigned to any node for execution. This also implies that it is the user who is responsible for predicting the execution characteristics (most notably runtime, and if applicable, data sizes). Furthermore, while most of the discussed works do consider the latency introduced by limited bandwidth, to the best of our knowledge, none of them actually treats it as a first-class citizen (in the sense of bandwidth being a limited resource that is susceptible to congestion and must be reserved, and that existing reservations must be considered). Finally, all the presented approaches consider compute resources (i.e., Grid worker nodes) as resources that will be exclusively assigned to single jobs, whereas DWARFS allows for concurrent reservations of fractions of CPU capacity, effectively being the only system we know that allows arbitrary tradeoffs between execution time and cost.

An interesting approach which allows to provide QoS guarantees and Advance Reservations, while still leaving some flexibility for both users and providers, is presented in [NBB07]. The idea is that users initially reserve for an activity execution within a given time window (which is larger than the actual time required). If the provider agrees, he commits to being able to provide the requested service at some point during that time interval (at his discretion). Users then have the possibility to re-negotiate (narrow) these windows as the execution approaches. This gives providers some flexibility to re-arrange reservations, because they do not initially commit to concrete times, but only to time frames. The authors evaluate several algorithms for rescheduling and other factors, and generally observe a low (5%) to significant (25%) system utilization gain.

A somewhat related concept termed priority provision, which targets large-scale grid networks, is introduced in [SVF06]. According to the authors, it can be understood as "a promise [...] that a certain capability will be available at sometime in the future without assigning a specific node to the client."

The work presented in [CD11] focuses on partitioning and scheduling Grid workflows in the presence of storage constraints. The assumption is that individual Grid sites have limited storage capacity which may not be sufficient to hold all of the data, and thus the workflow must be split into several fragments that are executed at different sites. On the other hand, activities (jobs) running at the same site and requiring the same data can leverage data locality and do not need to ship it between sites.

In [YKB07], the authors propose a multi-objective (deadline/cost) workflow execution planning approach based on genetic algorithms, where the result of the optimization is a set of pareto-optimal solutions for the user to choose from. As every activity can be performed by any provider, the resulting chromosome representation is very simple and elegant. The approach does take into account the duration (and cost) required for computations and data transfers. However, it is unclear whether it takes into account existing reservations or bandwidth restrictions.

A novel polynomial-time (more specifically, $O(m * n^3)$, where m = number of compute nodes, n = number of activities in workflow) heuristic for multi-objective planning optimization is presented in [FPBF12]. The algorithm can find pareto-dominant solutions for four dimensions, namely runtime, cost, energy consumption, and reliability. The authors have evaluated the algorithm using two real-world Grid workflows for the Askalon system, and found it to outperform other heuristic approaches, including Genetic Algorithms.

7.3 Monitoring and Prediction

As we very briefly discussed in Section 4.5, an empiric possibility to determine and predict future behavior of operation invocations is to gather historical data by monitoring the executions that have already taken place. For this prototypical implementation we have adopted the simplest possible strategy, namely to assume that all invocations essentially behave identically. In the following, we will give a short overview on more elaborate methods.

The work presented in [PBYC13] focuses on the profiling and execution time prediction of multi-threaded applications. The authors present a profiler which can monitor, at the Operating System kernel level, the thread scheduling behavior of running programs. This allows to derive a characteristical profile for the program in question, which can then be used to predict the behavior of future invocations.

Similarly, in [ACS+12], the interactions between multiple different programs are investigated. The paper presents an approach to monitoring various characteristics (such as CPU usage or disk I/O) of different programs, and extrapolating how these different programs affect each other. The original goal is to determine good combinations of multiple instances of different programs, in order to maximize resource usage without overloading the system. In the DWARFS context, if one used a more involved resource model, such capabilities could also be useful for operation providers – for instance to make better predictions for potential future invocations, based on already reserved invocations.

Another interesting approach, described in [HXHLB12], employs machine learning techniques to model an algorithm's runtime in terms of problem-specific instance features. In short, this means that in some cases, it's possible to predict the resulting runtime of the algorithm, even for previously unseen input.

8

Conclusion and Outlook

8.1 Summary

In this thesis, we have presented our approach to enabling the predictable execution of Scientific Workflows by using Advance Resource Reservations. The approach is targeting Service Oriented Architectures, i.e., settings where the individual activities of the workflow are provided by multiple independent and physically distributed service providers. The orchestration of workflow instances is performed by multiple cooperating Workflow Engines, and predictability is achieved by determining and reserving the required resources in a planning phase before the actual execution.

In Chapter 2, we started by establishing a model to formalize the concepts required for such a system, encompassing definitions for a broad range of abstraction levels. The notions of resources, their limited capacity, and the need to allocate their usage are at the heart of the model; on top of these, more abstract terms were then introduced and defined, ultimately leading to high-level definitions for concepts such as workflow schedules.

After having introduced the formal model, we turned our interest to aspects concerning the implementation of such a system. In order to set up the resource reservations needed to execute a workflow, a planning phase is first required. In Chapter 3, we discussed our approach to this optimization problem, which takes into account user-defined QoS criteria and is based on a Genetic Algorithm. We discussed aspects such as the chromosome representation and fitness functions, as well as features related to the distributed orchestration, such as partitioning and data transfer strategies.

Chapter 4 focused on the actual (provider-side) enforcement of reservations, i.e., on how the system can ensure that resource reservations are abided to. The enforcement of CPU allocations was given particular attention, and its implementation using a fuzzy controller was described.

Subsequently, we showed the “big picture” of the entire DWARFS infrastructure in Chapter 5. We presented the various services as well as their interactions, and discussed noteworthy practical considerations concerning the software deployment and configuration, as well as implementation aspects.

In Chapter 6, we presented various evaluations of both the planning and enforcement components. The planning correctly adapts to the state of the infrastructure (i.e., existing reservations) and produces acceptable, though sometimes suboptimal, results. The enforcement yielded very good results, demonstrating that workflow execution is indeed able to very closely abide to the predictions.

In Chapter 7, we gave an overview of related work in the area of Scientific Workflow Systems in general, as well as particular aspects concerning planning and scheduling problems, and supervision and runtime prediction.

8.2 Directions for Future Work

During the course of this thesis, several areas have been identified as being interesting topics for future work.

The most prominent one might be the support for failure handling, which has been almost completely omitted for the time being. In this area alone, multiple possibilities come to mind:

- Planning could be extended to account for unexpected failures, for instance by making redundant "plan B" reservations to provide failover alternatives. In that vein, one could also give users the option to trade security (multiple alternatives) against cost (no alternatives).
- Another possibility to handle failures could be to trigger an automatic re-planning at runtime, in the hope of being able to eventually terminate successfully instead of having to completely abort the execution.
- A related, but somewhat less serious fault is the (provider-side) failure to abide to the requested reservations, thus delaying the execution beyond the reserved time. A natural and simple way to prevent such events has already been shortly discussed in Chapter 6 – namely, to slightly overestimate the required resource usages (or underestimate resources' capacities). Again, this could be user-configurable to allow for a tradeoff between security and cost.

Another aspect lays in the area of the enforcement, monitoring, and prediction. For now, in order to evaluate the general feasibility, we have limited ourselves to simple examples, such as effectively parameterless operations that are purely CPU-bound. Extending the support to – for instance – I/O-bound operations, or operations which behave differently based on input data characteristics is an interesting open question.

Finally, there are other minor annoyances with the current implementation. For instance, we have not yet fully understood what exactly causes the irregularities that appear during the planning (Section 6.1.2), or if it is possible to prevent Glassfish from buffering data to disk (Section 6.2.1).

Bibliography

[ABJ+04] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew B. Jones, Bertram Ludäscher, and Steve Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, pages 423–424. IEEE Computer Society, 2004.

[ACS+12] Danilo Ansaloni, Lydia Y. Chen, Evgenia Smirni, Akira Yokokawa, and Walter Binder. Find your best match: predicting performance of consolidated workloads. In David R. Kaeli, Jerry Rolia, Lizy K. John, and Diwakar Krishnamurthy, editors, *ICPE*, pages 243–244. ACM, 2012.

[AH95] K. J. AAström and T. Hägglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, Research Triangle Park, NC, 2 edition, 1995.

[B+01] Peter Brittenham et al. Understanding WSDL in a UDDI registry. http://www.ibm.com/developerworks/webservices/library/ws-wsdl/, 2001.

[CAA+07] Leonardo Candela, Fuat Akal, Henri Avancini, Donatella Castelli, Luigi Fusco, Veronica Guidetti, Christoph Langguth, Andrea Manzi, Pasquale Pagano, Heiko Schuldt, Manuele Simi, Michael Springmann, and Laura Cristiana Voicu. Diligent: integrating digital library and grid technologies for a new earth observation research infrastructure. *Int. J. on Digital Libraries*, 7(1-2):59–80, 2007.

[CD11] Weiwei Chen and Ewa Deelman. Partitioning and scheduling workflows across multiple sites with storage constraints. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *PPAM (2)*, volume 7204 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2011.

[CDPEV05] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS-aware service composition based on genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, 2005.

[DGR+05] Kelvin K. Droegemeier, Dennis Gannon, Daniel Reed, Beth Plale, Jay Alameda, Tom Baltzer, Keith Brewster, Richard Clark, Ben Domenico, Sara Graves, Everette Joseph, Donald Murray, Rahul Ramachandran, Mohan Ramamurthy, Lavanya Ramakrishnan, John A. Rushing, Daniel Weber, Robert Wilhelmson, Anne Wilson, Ming Xue, and Sepideh Yalda. Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather. *Computing in Science and Engg.*, 7(6):12–29, 2005.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[FJP+05] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Jr., and Hong Linh Truong. ASKALON: a tool set for cluster and Grid computing. *Concurrency - Practice and Experience*, 17(2-4):143–169, 2005.

[FPBF12] Hamid Mohammadi Fard, Radu Prodan, Juan Jose Durillo Barrionuevo, and Thomas Fahringer. A multi-objective approach for workflow scheduling in heterogeneous environments. In *CCGRID*, pages 300–309. IEEE, 2012.

[Glo89] Fred Glover. Tabu search. *Part I," ORSA Journal on Computing*, pages 190–206, 1989.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[GRH05] Yang Gao, Hongqiang Rong, and Joshua Zhexue Huang. Adaptive grid job scheduling with genetic algorithms. *Future Gener. Comput. Syst.*, 21(1):151–161, 2005.

[HAR94] E. S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 5(2):113–120, 1994.

[HXHLB12] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: The state of the art. *CoRR*, abs/1211.0906, 2012.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[Kha08] Rania Khalaf. *Supporting business process fragmentation while maintaining operational semantics : a BPEL perspective*. PhD thesis, Universität Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2008.

[KMB08] Hajo N. Krabbenhöft, Steffen Möller, and Daniel Bayer. Integrating arc grid middleware with taverna workflows. *Bioinformatics*, 24(9):1221–1222, 2008.

[LS10] Christoph Langguth and Heiko Schuldt. Optimizing Resource Allocation for Scientific Workflows using Advance Reservations. In *SSDBM*, pages 434–451, 2010.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[NBB07] Marco Aurélio Stelmar Netto, Kris Bubendorfer, and Rajkumar Buyya. Sla-based advance reservations with flexible and adaptive time qos parameters. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 119–131. Springer, 2007.

[OAF+04] Thomas Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[Oxf10] Oxford English Dictionary Online, 3rd edition. `http://www.oed.com/`, March 2010.

[PA04] Cesare Pautasso and Gustavo Alonso. Jopera: a toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce (IJEC)*, 9:107–141, Winter 2004/2005 2004.

[PA05] Cesare Pautasso and Gustavo Alonso. Flexible binding for reusable composition of web services. In *In Proc. of the 4th Workshop on Software Composition*, pages 151–166, 2005.

[PBYC13] Achille Peternier, Walter Binder, Akira Yokokawa, and Lydia Y. Chen. Parallelism profiling and wall-time prediction for multi-threaded applications. In Seetharami Seelam, Petr Tuma, Giuliano Casale, Tony Field, and José Nelson Amaral, editors, *ICPE*, pages 211–216. ACM, 2013.

[PF05] Radu Prodan and Thomas Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 687–694, New York, NY, USA, 2005. ACM.

[Pla07] Beth Plale. Workload Characterization and Analysis of Storage and Bandwidth Needs of LEAD Workspace. In *Linked Environments for Atmospheric Discovery*, 2007.

[Res13] Wikipedia, Definition of "Resource". `http://en.wikipedia.org/wiki/Resource`, December 2013.

[Ros04] Timothy J. Ross. *Fuzzy Logic with Engineering Applications*. John Wiley & Sons, August 2004.

[San13] Glenn Santos. SSD Ranking: The Fastest Solid State Drives. `http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives#pcie`, January 2013.

[SF05] Mumtaz Siddiqui and Thomas Fahringer. GridARM: Askalon's Grid Resource Management System. In Peter M.A. Sloot, Alfons G. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *Advances in Grid Computing - EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 122–131. Springer Berlin Heidelberg, 2005.

[SIY08] D Singh, AM Ibrahim, and T Yohanna. A systematization of fundamentals of multisets. *Lecturas Matematicas*, 29:33–48, 2008.

[SPG06] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Framework for Collecting Provenance in Data-Centric Scientific Workflows. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, 2006.

[STS+06] Christoph Schuler, Can Türker, Hans-Jörg Schek, Roger Weber, and Heiko Schuldt. Scalable peer-to-peer process management. *Int. J. of Business Process Integration and Management*, 1:129–142(14), 8 June 2006.

[SVF06] Mumtaz Siddiqui, Alex Villazón, and Thomas Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized QoS. In *In the 2006 ACM/IEEE Conference on Supercomputing SC—06*, pages 103–118, 2006.

[Thea] The Apache Software Foundation. Apache jUDDI. http://juddi.apache.org/.

[Theb] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[YB05] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *J. Grid Comput.*, 3(3-4):171–200, 2005.

[YKB07] Jia Yu, Michael Kirley, and Rajkumar Buyya. Multi-objective planning for workflow execution on grids. In *GRID*, pages 10–17. IEEE, 2007.

Index

Allocation, 14
Allocation addition, 18
Allocation canonicity, 15
Allocation Cost, 20
Allocation equivalence, 15
Allocation validity, 16

Co-allocation, 20
Co-Allocation Commitment, 31
Co-Allocation Constraint, 26
Co-allocation Constraint Compliance, 27
Co-Allocation Constraint Graph, 28
Co-Allocation Constraint Restriction, 27
Co-allocation Cost, 20
Co-Allocation Feasibility, 31

Data Attribute, 37
Data Characteristic, 37
Dependent Allocation Constraint, 25
Dependent Allocation Constraint Compliance, 25
Dependent Allocation Constraint Restriction, 25

Endpoint, 39

Invocation Characteristic, 41

Operation, 6
Operation Instance, 40

Reservation, 29
Reservation Cancellation, 31
Reservation Cost, 29
Reservation State, 30
Reservation State Validity, 30
Resource , 11
Resource Classes, 11
Resource Provider, 13
Resource Type, 11

Time-determining Allocation Constraint, 22
Time-Determining Allocation Constraint Augmentation, 24
Time-Determining Allocation Constraint Augmented Restriction, 24
Time-Determining Allocation Constraint Compliance, 22
Time-Determining Allocation Constraint Restriction, 24

Usage Block, 13
Usage Block Validity, 14

Workflow Activity Precedence, 8
Workflow Description, 7
Workflow Description Validity, 8
Workflow Engine, 39
Workflow Schedule, 43
Workflow Schedule Co-allocation Constraint Graph, 48
Workflow Schedule Validity, 49

Curriculum Vitae

Christoph Langguth

January 31, 1978	Born in Jena, German Democratic Republic Son of Dagmar and Burkhard Langguth Citizen of the Federal Republic of Germany
1984–1985	Otto-Grotewohl-Schule, Jena, GDR
1985–1986	Karl-Liebknecht-Schule, Jena, GDR
1986–1988	School at the Embassy of the GDR in Warsaw, Poland
1988–1990	School at the Embassy of the USSR in Warsaw, Poland
1990–1991	School at the Embassy of Germany in Warsaw, Poland
1991–1996	School at the Embassy of France in Warsaw, Poland; Baccalauréat Général Scientifique
1996–1997	École d'Architecture de Lyon, Lyon, France
1997–2004	Friedrich-Schiller-Universität Jena, Jena, Germany; Diplom-Informatiker (equ. Master of Science in Computer Science)
2005–2005	UMIT, Hall in Tirol, Austria
2005–2014	University of Basel, Basel, Switzerland