Eindhoven University of Technology

MASTER

Design and implementation of workflow support in FLUENT

Verkooijen, J.W.

Award date:
2014

[Link to publication](#)

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Design and implementation of workflow support in FLUENT

Master Thesis

Student	J.W. Verkooijen (0720521) Embedded Systems J.W.Verkooijen@student.tue.nl
Supervisor	Dr. R.H. Mak System Architecture and Networking R.H.Mak@tue.nl
Tutor	A. Kuzmanovska MSc System Architecture and Networking A.Kuzmanovska@tue.nl

26 May 2014

System Architecture and Networking
Eindhoven University of Technology
Department of Mathematics and Computer Science

Abstract

Workflows are used to effectively divide the work of large-scale applications over the available resources. To support the scale of these applications, many resources are needed in order to provide adequate performance. There is a large set of existing workflow frameworks to manage the workflows and achieve and efficient resource usage.

FLUENT is a resource-aware component-based framework that is developed by the System Architecture and Networking group at Eindhoven University of Technology. FLUENT is designed for application life cycle management and for resource management. Given the resource management capabilities of FLUENT it seems natural to investigate whether FLUENT can be used as a workflow framework, and therefore design and implement the functionalities for workflow support.

Table of contents

Introduction 7

1.1 Context 7

1.2 Problem statement 7

1.3 Project goals 8

1.4 Outline of the thesis 9

High-level review of the FLUENT architecture 10

2.1 Overview of the FLUENT architecture 10

2.2 Responsibilities and interactions 12

2.2.1 FLUENT Framework Manager 12

2.2.2 FLUENT Resource Manager 13

2.2.3 FLUENT Device Manager 13

2.2.4 FLUENT Application Orchestrator 14

2.2.5 FLUENT GUI Tool 15

Responsibilities and requirements for workflow support 16

3.1 Responsibilities and interactions 16

3.1.1 FLUENT Framework Manager 16

3.1.2 FLUENT Resource Manager 16

3.1.3 FLUENT Device Manager 17

3.1.4 FLUENT Application Orchestrator 17

3.1.5 FLUENT Global Repository and FLUENT Local Repository 17

3.2 Functional requirements 18

Workflow definition in FLUENT 19

4.1 Application description 19

4.2 Workflow properties 19

4.3 Workflow language 20

4.4 Simple workflow example 21

4.5 Implementation of simple workflow 22

Data structures for workflow responsibilities 25

5.1 Data structures of the Framework Manager 25

5.2 Data structures of the Resource Manager 26

5.3 Data structures of the Orchestrator ... 27
Implementation of the workflow framework ... 28
6.1 Alive messages mechanism ... 28
6.2 Simple workflow application ... 29
6.2.1 Component communication ... 30
6.2.2 Sorter function ... 31
6.3 Workflow application deployment ... 31
6.4 FLUENT Resource Manager ... 32
6.4.1 UDP connection with other entities ... 33
6.4.2 Resource capacities ... 33
6.4.3 Reservation of resource capacities ... 34
6.5 Workflow scheduling ... 34
6.5.1 Scheduling policies ... 35
6.5.2 Scheduling workflow tasks ... 36
6.5.3 Workflow task finalization ... 38
Performance evaluation ... 40
7.1 Experimental setup ... 40
7.2 Experiments ... 41
7.2.1 Execution time of simple workflow application ... 42
7.2.2 Workflow scheduler ... 42
7.3 Experimental results ... 43
Conclusion and evaluation ... 49
References ... 51
A: Experimental results ... 52

Chapter 1

Introduction

Today’s software systems are developing and growing very fast. The quality delivered by such a large software system depends on the resources that are available to the system. Because the software systems are growing very fast, the applications that can be performed on such a system are also growing and will become very large-scale. To maintain the quality of the software systems, good management of the large-scale applications is needed. A possible way to do this is by treating these applications as workflows.

1.1 Context

In terms of software, a workflow application can be seen as a sequence of processing steps that together form a large application. The processes of such a workflow can be any process that requires a series of steps that can be automated via software. A workflow can be seen as a set of independent tasks. It can effectively divide the workload of an application over the available resources.

1.2 Problem statement

Today’s large-scale applications have three important characteristics according to R.H. Mak [1]. “They are *interdisciplinary*: Their design relies on knowledge from a wide range of domains of expertise, that is difficult to cover by a single design team.” “They are *intrinsically distributed*: Control of the application occurs at a location that is geographically distant from sensors and/or actuators.” “They require *intricate resource management*: On the one hand, intelligent reaction to sensor input or data content may give rise to large fluctuations in resource demands of the application itself. On the other hand, the application may be operating in an environment where it has to compete for resources with other applications that execute simultaneously, and hence may be confronted with fluctuating resource availability.” These characteristics have consequences for the design and deployment of such applications. The interdisciplinary nature favors a design style where applications are built from predefined components developed by experts in the various disciplines. These components do not only encapsulate specialist domain knowledge, the also facilitate reuse. For reuse of components to be successful, it must be complemented with reuse of architectural principles that facilitate interoperability between such components. A variety of component frameworks has been developed for this, but most of them are not resource-aware and do not accommodate resource-dependent dynamic reconfiguration.

FLUENT, as described by B. Orlic, I. David, R.H. Mak, and J.J. Lukkien in [2] and [3], is a resource-aware component-based framework. The FLUENT framework is designed for dynamic application life

cycle management and for resource management. FLUENT arbitrates the resource usage when multiple applications are competing for the same resources. The FLUENT application life cycle distinguishes three main phases in which an application can be engaged. Figure 1 shows this application life cycle.

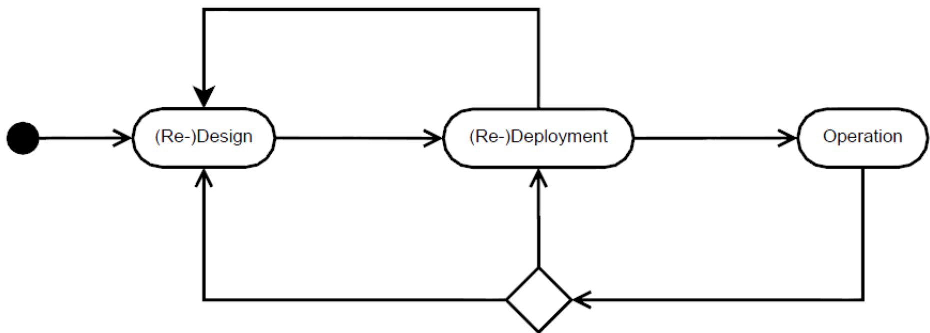

Figure 1: Application life cycle (taken from [1])

The most important phase in the life cycle is the operation phase. In this phase the application performs the services for which it has been designed. The (re-)design phase and the (re-)deployment phase represent deviations from this normal behavior during which the application is reconfigured. Reconfiguration is needed to resolve resource conflicts, and that is relevant for workflow applications. The reconfiguration phase consists of two distinct phases, redesign and redeployment.

The intention of FLUENT is to perform life cycle management for arbitrary service-oriented applications. Currently, the principal application domain supported by the framework is video processing.

Other application areas are involving large quantities of heterogeneous computational resources that require complex management, large data sets and large-scale scientific computing. As mentioned earlier, such applications are usually organized as a workflow application, and are executed on a Grid or in the Cloud. These computing infrastructures are equipped with management software that provides resource selection and allocation of workflows.

Given the resource management capabilities of FLUENT, it seems natural to investigate whether FLUENT can be used as a workflow management system, and thereby extending its current application domain.

1.3 Project goals

We want to extend the current FLUENT framework with support for workflow management. The goal of the project is to design and implement functionalities for this workflow support in FLUENT. The goal is realized by the following tasks:

- Review the current FLUENT architecture and describe the provided functionalities.

- Identify extensions and adaptations to the current FLUENT architecture necessary for workflow support.
- Design and implement these extensions.
- Select and execute a simple workflow.
- Evaluate the suitability of FLUENT for workflow management.

1.4 Outline of the thesis

This thesis is structured as follows. Chapter 2 reviews the current FLUENT architecture at a high-level. The chapter considers the responsibilities and interactions of the framework. The responsibilities, interactions and functional requirements for workflow management are considered in Chapter 3. Chapter 4 describes a definition of a workflow in FLUENT. Furthermore, it will define a simple workflow example, that will be used as a running example during the project for testing the implementation of the functionalities. In Chapter 5, the architecture of the framework is considered in more detail. It describes some design decisions that are made for workflow support. Chapter 6 describes the implementation of the workflow functionalities of the framework. In Chapter 7 the experimental setup and the experimental results are considered, and Chapter 8 concludes the Master project and gives an evaluation of the suitability of FLUENT for workflow management.

Chapter 2

High-level review of the FLUENT architecture

FLUENT is a distributed resource-aware component-based framework. It is developed for run-time composition of predictable multimedia processing applications. FLUENT looks for the best possible match between actual resource demand and resource availability. FLUENT is being designed and developed by the System Architecture and Networking (SAN) group at Eindhoven University of Technology. To enable FLUENT as a workflow framework it needs some extensions and adaptations that have been indicated during the pre-study of this Master project [5]. The following adaptations for FLUENT are indicated during the pre-study:

1. FLUENT needs a possibility to design and describe workflows.
2. The FLUENT Application Orchestrator needs adaptations to act as a workflow engine. It must be able to handle several policies and patterns.
3. The FLUENT Device Manager needs adaptations to have the ability to execute the workflows.
4. The FLUENT Resource Manager needs to contain all features necessary for discovering available and accessible resources for workflow execution.

The next sections give a high-level review the FLUENT architecture. An overview of the framework is given and its responsibilities are considered. A distinction is made between responsibilities of the framework that are currently present, and the ones that are needed for this assignment. The responsibilities needed for this assignment are considered in Chapter 3.

2.1 Overview of the FLUENT architecture

FLUENT consists of several components that are called the framework entities. The following framework entities are distinguished.

- A single FLUENT Framework Manager (FFM) entity that is deployed on the framework master node.
- A single FLUENT Resource Manager (FRM) entity that maintains an overview of available and allocated platform resources.
- A single FLUENT Global Repository (FGR) that is present at the framework master node.
- For each platform node exactly one FLUENT Device Manager (FDM) entity.
- For each platform node exactly one FLUENT Local Repository (FLR).
- A FLUENT Application Orchestrator (FAO) entity for every deployed component-based application.
- A FLUENT GUI Tool (FGT) entity for every user of the framework.

These framework entities are visualized in terms of a client-server architecture. The server side is divided into three layers, a master layer, an orchestrator layer, and a runtime support layer. The master layer consists of the FFM entity and the FGR, the orchestrator layer consists of a FAO entity, and the runtime support layer consists of a FDM and Dock entity and a FLR. The client side of the framework consists of FGT entities.

An overview of the abstractions of the FLUENT framework architecture is shown in Figure 2. The framework architecture consists of three layers, an application layer, a system layer, and a resource layer. The abstractions of the environment are shown at the left-hand side and the abstractions of the framework are shown at the right-hand side. The application layer contains an application orchestrator, that is responsible for the composition, deployment, and operation of an application. An application is composed of services which are provided by the components upon deployment. The system layer consists of a single framework manager that provides a registry-based entity subscription and entity discovery protocol, and a single resource manager that is in charge of all resources provided by the nodes of the distributed platform. To perform this task, the resource manager relies on services of local device managers. This is shown in the resource layer. The services include installation and instantiation of components, monitoring their resource usage, and enforcement of resource budgets. The system layer of the architecture also contains a repository that contains the service types and component types that are available in the system. Services are instances of services types and components are instances of component types.

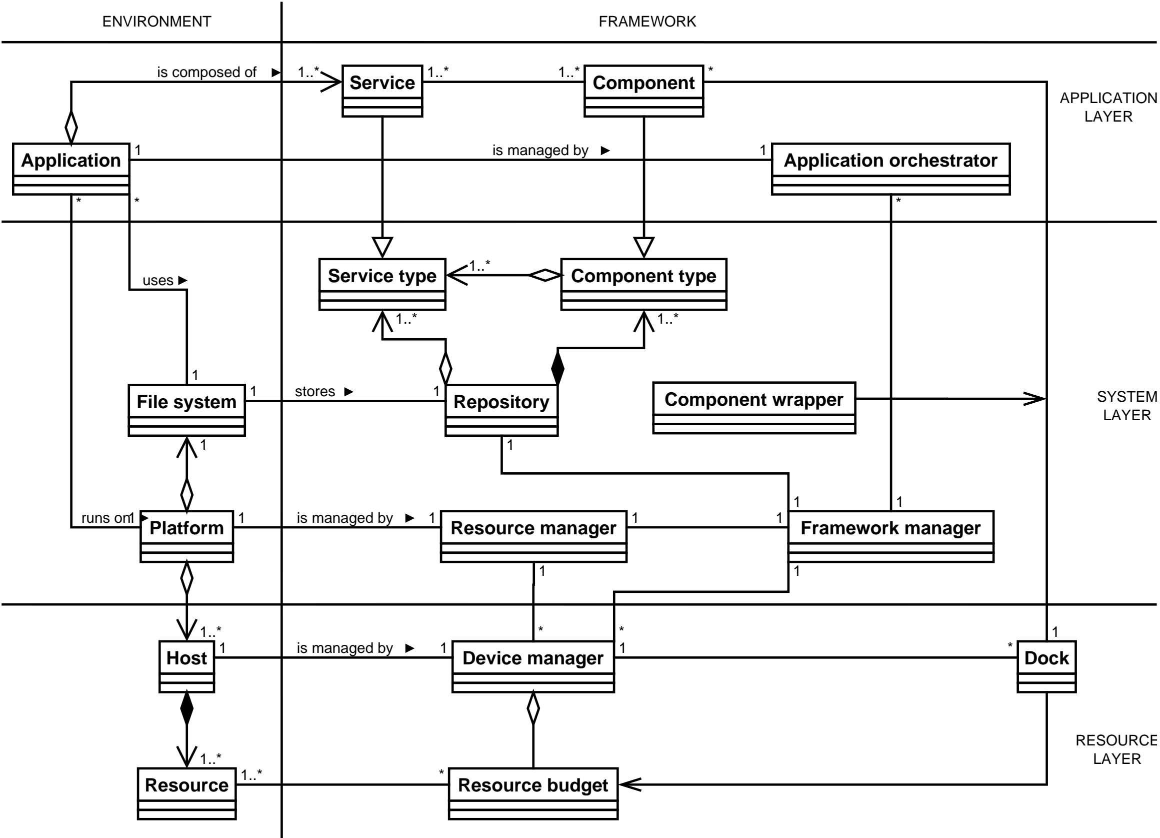

Figure 2: System overview

The overview of the system in Figure 2 does not contain the GUI Tool entity of FLUENT. The FGT provides a graphical user interface that allows a user to compose and deploy an application. The actual functionality for application composition and deployment is present on the FAO. The GUI Tool entities are the clients of the framework and they do not belong to the platform. That is why the GUI Tool is not contained in the system overview.

2.2 Responsibilities and interactions

One of the first implementations of FLUENT was made in the context of the ViCoMo [4] project, which is concerned with Video Context Modelling applications in the area of, for instance, surveillance and transport logistics. When starting with this project the application domain of the framework was video processing. Therefore we will use the term *streaming framework* in the remaining part of this thesis.

In the original design document of FLUENT that is created by SAN [6], there are some responsibilities and interactions for the framework. These responsibilities and interactions are considered in this section. Not all of them are present in the current implementation of the framework, and not all of them are relevant for this thesis.

2.2.1 FLUENT Framework Manager

The FFM is the central part of FLUENT. It has several important responsibilities in the current framework. When the framework starts up, one of the first tasks of the FFM is publishing its location to the outside world. By doing this, the FFM allows the other framework entities to find its location, and using its services. Further, the FFM maintains the framework model, i.e., all platform entities and the application entities deployed on them. Another important responsibility of the FFM is maintaining the framework's integrity. This integrity is captured in constraints such as: all framework entities have a unique identification, two Device Managers cannot reside on the same platform node, there is only one global repository (FGR) in the framework, and there is only one FRM. There is precisely one FAO per deployed application, and the global and local repositories are consistent. A final responsibility of the FFM is maintaining the framework user community, and providing access control. This last responsibility is not currently present in the framework.

Applications in FLUENT are realized using the Service-Oriented Architecture (SOA) style. An application in FLUENT is composed from different services. A FLUENT application is built from reusable and discoverable components that provide these services. A responsibility of the FFM is maintaining the framework applications. The FFM has responsibilities for opening an application and for closing an application. For opening an application the FFM has to fetch the application description from the FGR, including persistent data. It has to create a FAO that manages the application, and it has to add an entry in the application table that stores vital information regarding the application, such as the identity and location of its Orchestrator and its application status. For closing an application the FFM has to store data that has to preserved between successive application sessions. It has to remove the application entry from the application table. It has to destroy the FAO that managed the application, and it has to update the application status.

Finally, the FFM assists the various framework users in setting up and initializing platform entities, such as the installation of the FRM and FDMs. Furthermore, the FFM issues FDMs to install an initial component distribution, i.e., preloaded plugin libraries, support libraries, and libraries containing application specific plugins.

To fulfill its responsibilities the FFM interacts with several framework entities and components. The FFM interacts with all framework entities to keep track of their location and alive status. The FFM interacts with the environment to publish its location, and it interacts with the FDMs for installation purposes. There is also an interaction with the FGR, to maintain a persistent copy of the framework model, and a persistent copy of the submitted applications. There is an interaction with the FAOs for maintaining the application life cycle status, and finally an interaction with the FGT to update, store, and submit application descriptions.

2.2.2 FLUENT Resource Manager

The FLUENT *streaming framework* has no implementation of the FRM. However, the FRM is specified in the original FLUENT design document of SAN. Because resource discovery and resource management are an important part of a workflow framework, FLUENT will need the FRM in the implementation that has to support workflows. The responsibilities and the interactions of the FRM of that implementation are considered in Chapter 3.

2.2.3 FLUENT Device Manager

The framework has one or more worker nodes associated to it. Every worker node consists of a single FDM entity that manages one or more Dock entities according to the requests of the application deployment configuration. Dock entities can host multiple components from the same applications, and they manage the connections between application components hosted by them. The FDM has the following responsibilities in the current framework.

The FDM discovers the service with the location of the FFM, and subscribes itself when it starts up. This service discovery is one of the deliverables of the pre-study. A FDM installs and uninstalls components, i.e., it downloads component types, it registers the installed component types at its local repository (FLR), and it keeps track of its location for each installed component type. Another responsibility of the FDM is deploying and controlling components. It keeps track of the number and identity of active components of each installed component type. It maintains the control status for each component. It creates access points for each service of each active component and registers these at the appropriate FAOs, and it notifies the FAOs of changes of the control status.

Further, the FDM provides an operator interface for monitoring purposes and configuration purposes.

To fulfill its responsibilities a FDM interacts with the following entities and components. The FDM interacts with the FFM to obtain its location, and to subscribe itself. The FDM interacts with the FGR to upload component types for installation, and register installed component types. There is an interaction with the FAOs to register service access points, perform application composition, signal mismatches between resource usage and resource budget, and signal error conditions. The FDM

interacts with the FRM to report node resource profiles, and upload component resource reservations. Finally, it interacts with third-party users to install component types, and deploy components.

2.2.4 FLUENT Application Orchestrator

The FAO provides functionality to configure, deploy, and monitor an application. The FAO acts as an application manager and it has the following responsibilities in the current framework.

A FAO discovers the service of the FFM location, and it subscribes itself at the FFM when it starts up. The FAO is started by the FFM when an application has to be opened. A FAO manages the life cycle of its application, and it keeps the FFM informed about the status of an application. A FAO is able to manage only one application at a time. A FAO determines a mapping of application components to worker nodes.

Further, a FAO supervises the deployment of the application by the FDMs in accordance with the allocation. To that end, it instructs the FDMs to ensure the presentence of the required component types, it instructs the FDMs to create the required Docks including component instantiation, it informs the instantiated components of the required bindings, and it instructs the Docks to performs life cycle management commands on the individual components. Figure 3 shows the activity diagram of the application life cycle. The boxes in the diagram denote the activities in the life cycle of an application. The status of the application is implicit on the arrows in between the activities.

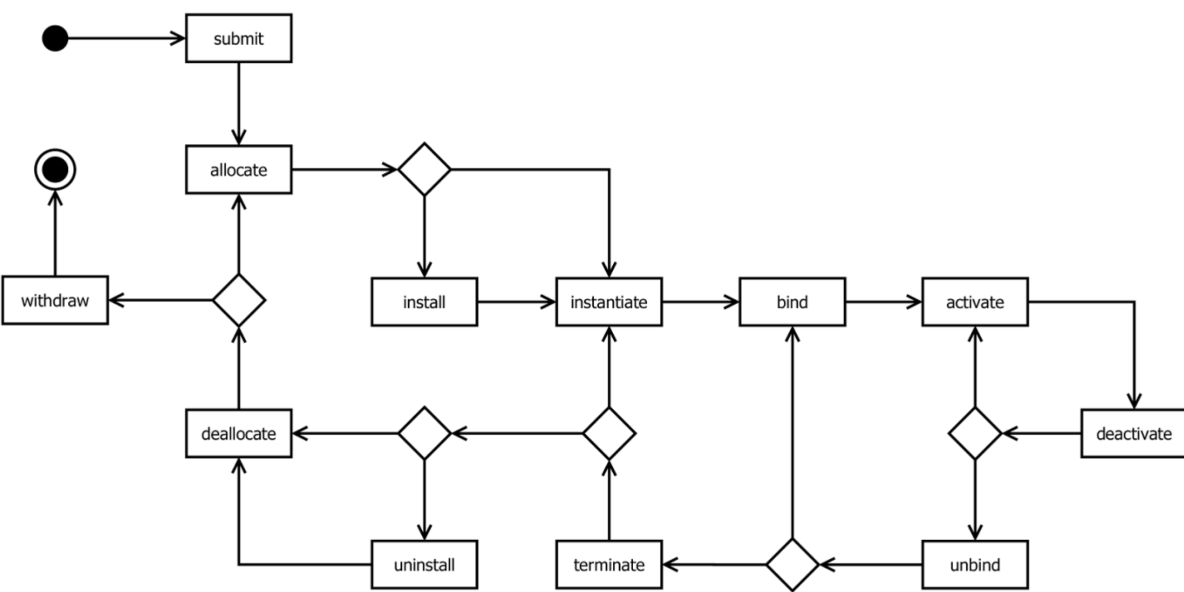

Figure 3: Activity diagram of the application life cycle

To fulfill its responsibilities the FAO has to interact with the following entities and components. The FAO needs to interact with the FFM to obtain its location, to subscribe itself, and inform it of the status of the managed application. It needs an interaction with the FDMs and with the Docks to

inform them about the application deployment. Finally there must be an interaction between the FAO and the components of its application to inform them of the required bindings.

2.2.5 FLUENT GUI Tool

The client side of the framework consists of FGT entities. The FGT has the following responsibilities in the current framework.

A FGT discovers the service of the FFM location, and it subscribes itself at the FFM when it starts up. A FGT offers a set of interfaces for application management. It can discover the available components, it can compose applications, it can deploy composed applications, it can monitor an application, and it can reconfigure applications. For the deployment of an application, the FGT requests the FFM for an available FAO and it connects to that FAO. The number of client nodes is determined by the number of applications running at the same time. A single FGT entity can serve only one running application at a time.

To fulfill its responsibilities the FGT has to interact with the following entities. The FGT has to interact with the FFM to obtain its location, to subscribe itself, and to request for an available FAO. Further, the FAO has to interact with the FAO to inform it about the deployment of the application and about the bindings between the components of the application.

Chapter 3

Responsibilities and requirements for workflow support

In Section 2.2 the responsibilities and interactions of the framework entities in the *streaming framework* are described. The framework entities need additional responsibilities and interactions when we want FLUENT to be able to support workflows. The term *workflow framework* will be used in the remaining part of this thesis when talking about FLUENT with adaptations for workflow support.

3.1 Responsibilities and interactions

This section described the additional responsibilities and interactions for the FLUENT *workflow framework*.

3.1.1 FLUENT Framework Manager

The FFM does not need large changes and additional functionalities to use FLUENT as a workflow framework. However, there are some small changes. Therefore the FFM has additional responsibilities in the *workflow framework*. There are some differences between FLUENT applications and workflow applications, such as tasks of a workflow application are executed once, while FLUENT applications run indefinitely. Because these differences, the FFM also has to maintain the workflow applications. It has to fetch the workflow application description from the FGR, it has to provide information to the FAO that manages the workflow application, and it has to add an entry in the application table that stores vital information regarding the application. A final responsibility for the FFM in the *workflow framework* is to check the alive status of the other framework entities. The other framework entities have to send every alive period a message to the FFM to inform the FFM about their alive status. This alive message mechanism is described in section 6.1.

3.1.2 FLUENT Resource Manager

The FRM is one of the important parts of the FLUENT *workflow framework*. It manages all resources in the framework that are needed for the scheduling of a workflow. The FRM is not present in the *streaming framework* and it has to be implemented for the workflow support in FLUENT. The FRM needs the following responsibilities in the *workflow framework*. The FRM maintains an overview of the available and allocated platform resources. It negotiates budget reservations with FAOs. The FRM is responsible for arbitration of conflicting resource demands between applications. The FRM has to issue the deployment of components on platform nodes. The FRM has to collect resource usage data from the FDMs, and the FRM has to provide this resource usage data to operators.

To fulfill its responsibilities the FRM has to interact with the following entities and components. The FRM has to interact with the FDMs to obtain resource usage data. This data is aggregated over all applications that are deployed on the FDMs host. Further, the FRM has to interact with the FDMs to issue resource reservations per application. Finally, there has to be an interaction with the FAOs to negotiate resource reservations. The negotiation process is such that it guarantees that for any host the reserved budgets do not exceed the nodes capacity.

3.1.3 FLUENT Device Manager

For the *workflow framework*, the FDM will be used for the execution of workflow tasks. Because there are some differences between FLUENT applications and workflow applications, there are also some extensions to the responsibilities of the FDM. The FDM must be able to execute one or more tasks of a FLUENT workflow. To fulfill this responsibility, the FDM has to interact with the FRM to report resource usage data, and with the FAO to receive instructions about workflow execution.

3.1.4 FLUENT Application Orchestrator

The FAO is the most important part of the *workflow framework*, and it is special to the application domain. The FAO has some important responsibilities for workflow execution and for scheduling of a workflow.

A FAO must be able to read and translate a workflow application description. A FAO must be able to handle several workflow policies and patterns. Another responsibility of the FAO in the *workflow framework* is that it must be able to schedule the tasks of a workflow over the available and accessible resources. To obtain these resources reservations, the FAO has to negotiate with the FRM. There is interaction with the FRM for obtaining the available resources, for reservations of the resources, and for the release of the resource reservations. The FAO keeps the FFM informed about the status of the workflow application. A final responsibility of the FAO is that it has to instruct the FDMs to execute a task or multiple tasks of the FLUENT workflow.

3.1.5 FLUENT Global Repository and FLUENT Local Repository

FLUENT is developed for video processing applications. Therefore, the communications between different components of an application is done by streaming, and there is no communication via files.

The *workflow framework* makes use of file communication. The components of a workflow application may use some files that are needed for their execution. This files are located in the repositories of the framework. The responsibility of the repositories is to store the files that are needed for a certain workflow application. The file communication between workflow components is explained in Section 6.2.1.

3.2 Functional requirements

There can be some important functional requirements derived using the responsibilities and interactions of the *workflow framework*. Requirements for the *workflow framework* are listed below.

- When a user submits an application, one of the responsibilities of the FFM is to start a new FAO to manage that application. In that case there is a new FAO child process started from the FFM parent process, and the terminal output of the new process is not visible inside the parent process. For workflows, the FAO is an important framework entity. Therefore it is required for this project that the FAO can be started manually as a separate process by the user, such that all output is visible, like scheduling decisions and task executions.
- Every worker node in the system needs a fixed configurable percentage of free CPU and RAM capacity. These capacities are used for testing and visualizing the workflow scheduler on the FAO.
- A user of the framework must be able to set the percentage of capacity that is needed for a workflow task to execute.
- A user of the framework must be able to choose a scheduling policy that is used by the workflow scheduler on the FAO.
- There has to be a file system that is used to control the data that is stored and retrieved by a workflow application.

Chapter 4

Workflow definition in FLUENT

To use FLUENT as a workflow framework, a workflow has to be represented in some predefined format. FLUENT must be able to read and understand this workflow format, such that it can handle the workflow.

4.1 Application description

An application in FLUENT is described by an application description. Essentially, a FLUENT application description contains a set of participating component instances, a set of intended interface bindings between them, and a set of application configurations. To create such an application description, the FGT entity is needed. Using the FGT, the application can be configured and the bindings between the components of the application can be set.

4.2 Workflow properties

The FGT entity is used to describe an application in the FLUENT *streaming framework*. In the *workflow framework* the FGT entity is not used anymore. To allow a framework client to create a workflow application, there needs to be a workflow definition. A workflow definition in FLUENT needs to contain the following properties:

- A workflow contains a name.
- A workflow contains a description that describes what the workflow should do.
- A workflow contains a finite set of tasks $\mathcal{T}$.
- A workflow contains a finite set of sources $\mathcal{S}$. This are tasks $t \in \mathcal{T}$ which can start immediately.
- A workflow contains a finite set of sinks $\mathcal{Z}$. This are tasks $t \in \mathcal{T}$ that are the final tasks that can be executed for the workflow.
- Every task $t \in \mathcal{T}$ contains a component $\mathcal{C}$ which has to be executed for the task execution.
- Every task $t \in \mathcal{T}$ contains a finite set of files $\mathcal{F}_{in}$ that is needed to perform the task.
- Every task $t \in \mathcal{T}$ contains a finite set of files $\mathcal{F}_{out}$ that is produces by performing the task.
- Every task $t \in \mathcal{T}$ contains a finite set of dependencies $\mathcal{D}$. If a task $t_2 \in \mathcal{T}$ depends on another task $t_1 \in \mathcal{T}$, task t_2 can only be executed if task t_1 has been executed.

4.3 Workflow language

Any workflow framework needs a workflow language to describe workflows. To allow FLUENT to read and translate a workflow, it also needs a workflow language that must be written in a predefined format. There are several existing languages that can help with the description of a workflow, such as XML [7] and JSON [8]. XML (Extensible Markup Language) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. JSON (JavaScript Object Notation) is an open standard format that uses human-readable text to transmit data objects. JSON is primarily used to transmit data between a server and a web application, as an alternative to XML. Another possibility for a workflow language is to create a whole new language for FLUENT. A strong point of writing a new language is that weaknesses of existing languages can be tempered for usage in FLUENT. A new language will be specific for and only readable by FLUENT. This can be seen as a weakness because it decreases the usability of the language. There are existing tools for monitoring, debugging and other necessary tasks for existing languages. For a new language there are no such tools and the users must be real experts to work with it.

FLUENT makes use of the POCO C++ Libraries [9]. The POCO C++ Libraries are a collection of open-source C++ class libraries written in efficient, modern ANSI/ISO Standard C++ and based on the C++ Standard Library/STL. They are developed for building network- and internet-based applications. One of the features of POCO is XML parsing (SAX2 and DOM) and XML generation. POCO has currently no support for the JSON format.

Another argument for using XML is that it is already used in the current FLUENT framework for serialization. Considering these arguments, XML is chosen as the workflow language for FLUENT. The following example shows the XML representation of a workflow in FLUENT that has three tasks.

```
<?xml version="1.0" encoding="utf-8"?>
<workflow name="NAME OF THE WORKFLOW">
 <description>DESCRIPTION OF THE WORKLFOW</description>
 <sources>
  <source>TASK1</source>
 </sources>
 <sinks>
  <sink>TASK3</sink>
 </sinks>
 <tasks>
  <task name="TASK1">
   <component name="COMPONENT1" version="VERSION">
    <description>DESCRIPTION OF COMPONENT1</description>
   </component>
   <inputfiles>
    <file>FILE1</file>
   </inputfiles>
   <outputfiles>
    <file>FILE2</file>
   </outputfiles>
   <dependencies/>
  </task>
  <task name="TASK2">
   <component name="COMPONENT2" version="VERSION">
```

```
     <description>DESCRIPTION OF COMPONENT2</description>
    </component>
    <inputfiles>
     <file>FILE2</file>
    </inputfiles>
    <outputfiles>
     <file>FILE3</file>
    </outputfiles>
    <dependencies>
     <depend>TASK1</depend>
    </dependencies>
   </task>
   <task name="TASK3">
    <component name="COMPONENT3" version="VERSION">
     <description>DESCRIPTION OF COMPONENT3</description>
    </component>
    <inputfiles>
     <file>FILE3</file>
    </inputfiles>
    <outputfiles/>
    <dependencies>
     <depend>TASK2</depend>
    </dependencies>
   </task>
  </tasks>
</workflow>
```

4.4 Simple workflow example

A simple synthetic workflow example is needed to test all implementations of the *workflow framework* and all other changes to FLUENT. There are some requirements for such a workflow example. The workflow needs to be easily configurable to create a variety of workloads, and it has to contain a number of distinct components which various workflows can be build. To create such a simple workflow a general problem is formulated as follows.

We want to know the possibility that a random generated set of natural numbers is a subset of a larger set of natural numbers. A possible way to solve this problem is described by the following definition.

Let $\mathcal{U} = \{i | 0 < i < 2^{64}\}$ be a universe of natural numbers. Using this universe there is a set with random generated numbers $U \subseteq \mathcal{U}$ where $|U| = 2^u$ and $15 \leq u \leq 19$, and there are sets with random generated numbers $X_i \subseteq \mathcal{U}$ where $|X_i| = 2^x$, $10 \leq x \leq 14$ and $1 \leq i \leq n$.

To allow an easier and faster intersection of these sets, the sets of random generated numbers are sorted in ascending order. It is also possible to calculate the intersection of the sets without sorting them first. Using the ordered sets the intersection of sets U and X_i is calculated: $Y_i = U \cap X_i$. Using the sets Y_i the average length is calculated: $z = \frac{1}{n}\sum_{1 \leq i \leq n} y_i$ where $y_i = |Y_i|$. A diagram of the simple workflow is showed in Figure 4.

This simple workflow is only there to create the possibility to test a variety of workloads. However the workflow is configurable, it does not generate any useful output. By using the sorting functions, additional workflow tasks are created and they generate additional workload.

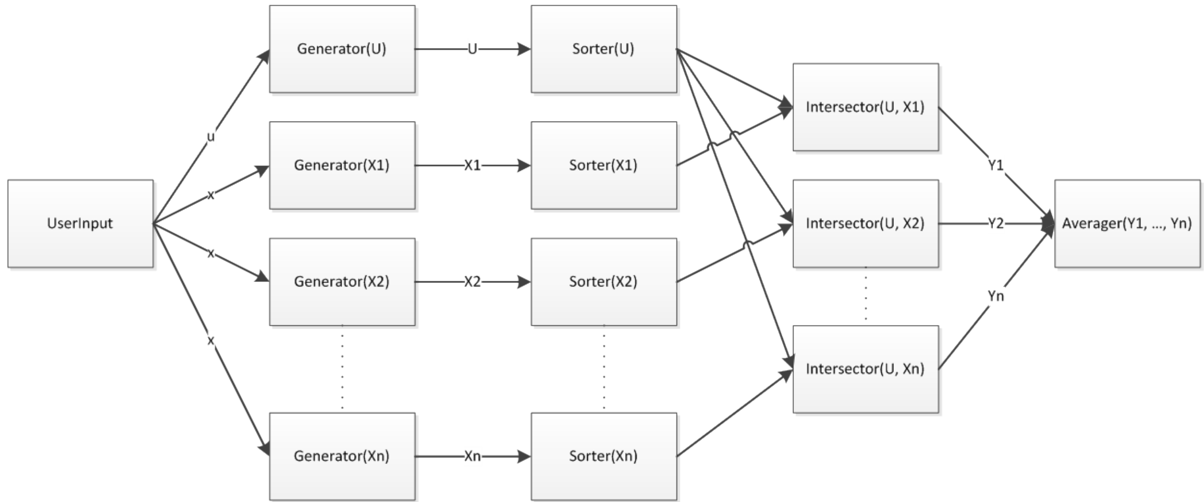

Figure 4: Simple workflow example

4.5 Implementation of simple workflow

The simple workflow that is described in paragraph 4.4 is implemented in the workflow language that is described in paragraph 4.3. The following example shows the XML representation of the simple workflow where $n = 1$.

```
<?xml version="1.0" encoding="utf-8"?>
<workflow name="Simple_Workflow">
 <description>
  This workflow calculates the average length of the
  intersection of several sets
 </description>
 <sources>
  <source>T1</source>
</sources>
 <sinks>
  <sink>T7</sink>
 </sinks>
 <tasks>
  <task name="T1">
   <component name="UserInput" version="">
    <description>Sends set length to T2 and T3</description>
   </component>
   <inputfiles/>
   <outputfiles/>
   <dependencies/>
  </task>
  <task name="T2">
   <component name="Generator(U)" version="">
    <description>Generate random numbers U</description>
   </component>
   <inputfiles/>
   <outputfiles>
```

```
  <file>randomNumbersU</file>
 </outputfiles>
 <dependencies>T1</dependencies>
</task>
<task name="T3">
 <component name="Generator(X1)" version="">
  <description>Generate random numbers X1</description>
 </component>
 <inputfiles/>
 <outputfiles>
  <file>randomNumbersX1</file>
 </outputfiles>
 <dependencies>T1</dependencies>
</task>
<task name="T4">
 <component name="Sorter(U)" version="">
  <description>Sort random numbers U</description>
 </component>
 <inputfiles>
  <file>randomNumbersU</file>
 </inputfiles>
 <outputfiles>
  <file>sortedNumbersU</file>
 </outputfiles>
 <dependencies>
  <depend>T2</depend>
 </dependencies>
</task>
<task name="T5">
 <component name="Sorter(X1)" version="">
  <description>Sort random numbers X1</description>
 </component>
 <inputfiles>
  <file>randomNumbersX1</file>
 </inputfiles>
 <outputfiles>
  <file>sortedNumbersX1</file>
 </outputfiles>
 <dependencies>
  <depend>T3</depend>
 </dependencies>
</task>
<task name="T6">
 <component name="Intersector(U, X1)" version="">
  <description>Intersection of U and X1</description>
 </component>
 <inputfiles>
  <file>sortedNumbersU</file>
  <file>sortedNumbersX1</file>
 </inputfiles>
 <outputfiles>
  <file>intersectionNumbersY1</file>
 </outputfiles>
 <dependencies>
  <depend>T4</depend>
  <depend>T5</depend>
 </dependencies>
</task>
<task name="T7">
 <component name="Averager()" version="">
```

```
    <description>Calculate average length of Y1</description>
   </component>
   <inputfiles>
    <file>intersectionNumbersY1</file>
   </inputfiles>
   <outputfiles>
    <file>averageZ</file>
   </outputfiles>
   <dependencies>
    <depend>T6</depend>
   </dependencies>
  </task>
 </tasks>
</workflow>
```

Chapter 5

Data structures for workflow responsibilities

The FLUENT framework maintains state, i.e., information about its own configuration and resources, and for the life cycle management of applications. Some are specific for the workflow domain, but not all, e.g., the resources are independent of the applications executed. The framework entities contain several data structures that help the framework with storing and organizing the data for state maintenance in an efficient way. The important data structures that are used in the (workflow) framework are considered in this chapter.

5.1 Data structures of the Framework Manager

The FFM contains a data structure for all platform entities in the framework. The *streaming framework* makes use of a fixed platform for video processing applications. In case of the *workflow framework* the platform may grow and shrink dynamically. Therefore, the framework has to be aware of the entities that are currently present in the framework.

The *Platform table* contains the following fields for every entity that is subscribed to the framework:

- Entity id
- Entity host name
- Entity host address
- Entity type
- Entity status

In the Platform table there is an entry for every subscribed entity with the id of that entity, the name of the host of the entity, the address of the host, and the type of entity. Further there is a field that contains the status of the entity, where status ∈ {*ALIVE*, *NOT_ALIVE*}. The mechanism that determines the status of an entity is described in Section 6.1.

One of the responsibilities when starting the FFM is to create the Resource Manager. Another responsibility is to create FAO entities for managing submitted applications. To start these entities, the FFM has to create a new child process from its parent process. To avoid segmentation faults and other errors when shutting down the framework, all child processes must be terminated correctly by its parent process. Therefore, the FFM contains a *Process table* for every process started from the FFM process:

- Entity id
- Process id
- Process handle

Every process entry in the table has a unique id of the entity that is associated with the process, it contains the id of the process and contains the handle of the process. The process handle is needed to communicate with the child processes that are created by the FFM.

The FFM runs a separate thread that checks the alive status of all entities subscribed to the framework. Every entity must send every alive period an alive message to the FFM. The FFM contains an *Alive status table* for all subscribed entities to the framework.

Alive status table, for every entity subscribed to the framework:

- Entity id
- Creation timestamp
- Latest timestamp
- Alive period

Every entry in the table contains the id of the subscribed entity, the timestamp of the time that the entity is created, the timestamp of the latest time that the entity has sent a message to the FFM, and the alive period of the entity. The FFM uses the alive period of every entity to decide whether the status of the entity is *ALIVE* or *NOT_ALIVE*.

A last important data structure on the FFM is the *Application table* for managing all applications.

Application table, for every application in the framework:

- Application id
- Application name
- Application location
- Application status
- A reference to the application descriptor

For every application in the Application table there is a unique id and a name of the application. The application location contains the path where the application is located. Every application has a status, where status ∈ {*SUBMITTED*, *ALLOCATED*, *INSTALLED*, *INSTANTIATED*, *BONDED*, *ACTIVATED*}. These statuses are derived from the application life cycle that is shown in Figure 3. The last field in the table is a reference to the application descriptor that describes the application such that FLUENT knows how to execute it.

5.2 Data structures of the Resource Manager

The FRM contains a data structure with capacities of all worker nodes in the framework. When a worker node subscribes to the framework, the FDM connects to the FRM, and the FRM adds a new entry to its *Capacity table*. The connection that is made between the FDM and the FRM is explained in Section 6.4.

Capacity table, for every FDM subscribed to the framework:

- FDM id
- Free CPU capacity
- Reserved CPU capacity
- Free RAM capacity
- Reserved RAM capacity

Every worker node in the Capacity table has a unique id. Every entry in the table contains a field for the free CPU capacity, a field for the reserved CPU capacity, a field for the free RAM capacity and a field for the reserved RAM capacity. The free capacities are available for new applications, and the reserved capacities are used by active applications.

5.3 Data structures of the Orchestrator

The FAO contains a data structure with all available worker nodes that have at least a certain capacity. When the FAO has to schedule a workflow, it requests the FRM for the resources that satisfy a certain minimum capacity to schedule the workflow. These resources are stored in the local *Worker nodes table* of the FAO. These resources are stored locally to avoid sending messages to the FRM and waiting for their replies during the scheduling process. The Worker nodes table is periodically updated by sending a request to the FRM to avoid inconsistency.

Worker nodes table:

- FDM id
- Percentage of free CPU capacity
- Percentage of free RAM capacity

The Orchestrator uses this table for scheduling. It has to decide where to schedule a certain task of a workflow. Therefore it uses the resources in the Worker nodes table. When the FAO uses a resource from its table to schedule a workflow task, it negotiates with the FRM for a reservation of a certain capacity of a resource. The FRM updates its Capacity table. The FAO has to store the information of the scheduled components. Therefore the FAO uses the *Workflow components table* with all scheduled components of the workflow application.

Workflow components table:

- Component name
- Percentage of reserved CPU capacity
- Percentage of reserved RAM capacity
- Scheduled node

The Orchestrator uses this table to manage resource reservations for all tasks of a workflow. For all tasks in this table there is an entry with the name of the task, the percentage of reserved CPU capacity, the percentage of reserved RAM capacity, and the id of the node on which the node is scheduled. When tasks of a workflow are completed, the FAO knows that the reserved capacities of the scheduled node can be released.

Chapter 6

Implementation of the workflow framework

This chapter describes the implementation of the *workflow framework*. That means that it describes the adaptations and additions that are made to the FLUENT *streaming framework* to enable workflow support.

6.1 Alive messages mechanism

One of the first adaptations of the FLUENT framework was the implementation of an alive messages mechanism between the FFM and the other framework entities. The alive messages mechanism is independent of workflow, but it is important. The *streaming framework* was developed for video streaming applications, and uses a fixed platform. The *workflow framework* uses a dynamic platform, i.e., framework entities can subscribe to the platform, but they can also disappear.

When a framework entity starts up and subscribes itself at the Framework Manager it makes the FFM aware of its alive period. The entity decides its alive period at its initialization phase. Figure 5 shows a sequence diagram of alive messages that are sent between a FDM and the FFM.

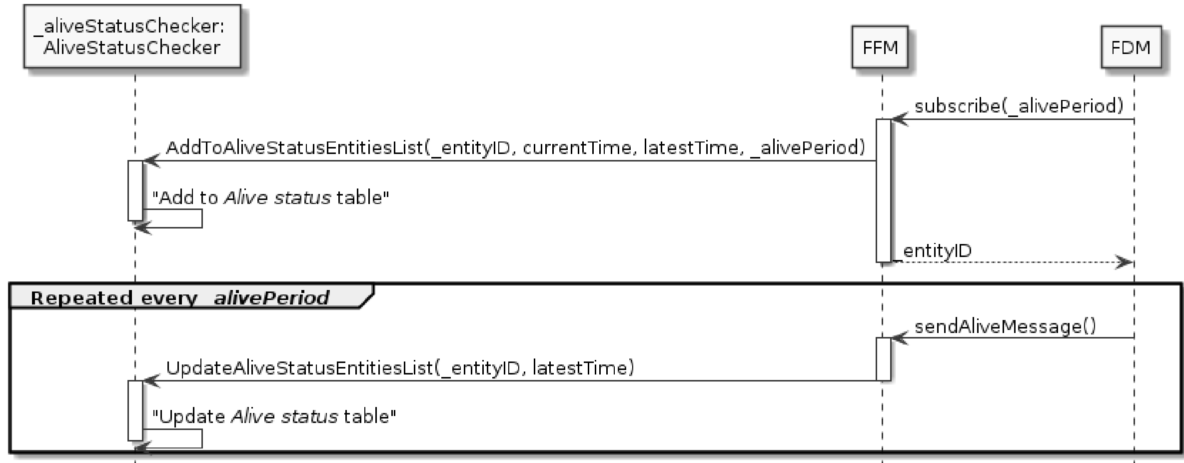

Figure 5: Sequence diagram of alive messages that are sent between FDM and FFM

The FFM contains a class AliveStatusChecker that runs on a separate thread. Inside that class there is an Alive status table with entries for all subscribed entities. The Alive status table is considered in Section 5.1. The AliveStatusChecker class checks every check period the entries in this table and decides according to the timestamp of the latest alive time whether the entity is alive or not. Note

that the check period must be smaller than the smallest alive period of all subscribed entities, to avoid missing of alive messages. Every subscribed entity that is alive sends every alive period an alive message to the Framework Manager with a timestamp of its latest alive time. The FFM updates the latest alive timestamp in its Alive status table. If the FFM receives four times no alive message from an entity, the status of that entity is changed to *NOT_ALIVE* and the entity is removed from the Alive status table. The FFM checks for every subscribed entity the following condition to decide whether an entity is alive or not:

$$t_{now} > \ t_{last} + c.p_{alive}$$

where t_{now} is the current timestamp, t_{last} is the timestamp of the latest alive time of the entity, c is a condition parameter, and p_{alive} is the alive period of the entity. The condition parameter is in this case $c = 4$. When the condition holds, the FFM knows that the entity is not alive anymore. Figure 6 shows a sequence diagram of the check mechanism of the alive status of the entities.

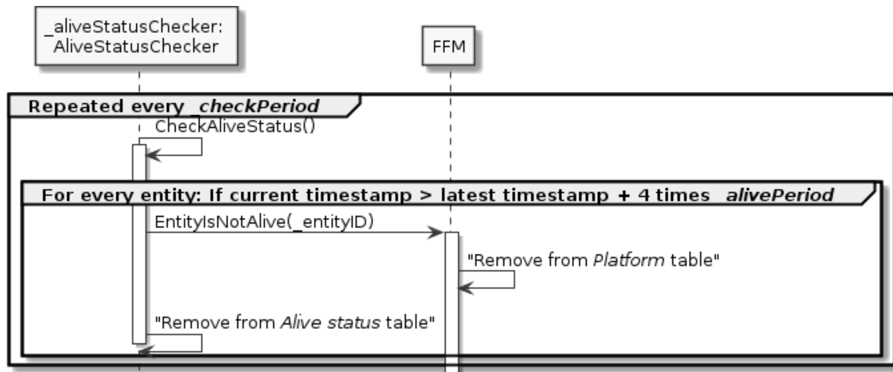

Figure 6: Sequence diagram of checking the alive status of the entities

For workflows it is important that the resources are available and when they are not alive and thus not available anymore, that the framework is aware of this. That is why the alive messages mechanism is important for the *workflow framework*.

6.2 Simple workflow application

As described in [5], FLUENT is a framework that supports component-based applications. FLUENT applications are composed from independent components with well-defined interfaces and specified behavior. As can be seen from Figure 4, the simple workflow application has multiple workflow tasks. Each individual task can be seen as a software component. The workflow application needs a component *UserInput*, a component *Generator*, a component *Sorter*, a component *Intersecter*, and a component *Averager*. The *UserInput* component adds no additional functionality to the workflow, but it is used to send the numbers u over the channel to the Generator(U) and a number x to the Generator(X_i), where $1 \leq i \leq n$. According to the value of the number that is received at its required interface, the Generator components know whether they has to act as a Generator(U) or as a Generator(X_i), i.e., they know the size of the set that has to be generated. In

between the Sorter(U) task and the *Intersecter* components there is an additional component called *Demux*. This component has one input and two outputs and it is used to send the output of the sorter to both the *Intersecter* components. It is not possible in FLUENT to connect an output channel directly to two input channels.

6.2.1 Component communication

In the *streaming framework*, components in FLUENT communicate by streaming, they are sending data over channels and they do not store any data in files. The simple workflow application makes use of files to store the large sets of numbers generated by the application. The components send the location of the output files over the channel to the next component. The next component knows where the file with its input data is located, and reads the file to obtain all data needed to perform its task. Figure 7 shows the communication between the components.

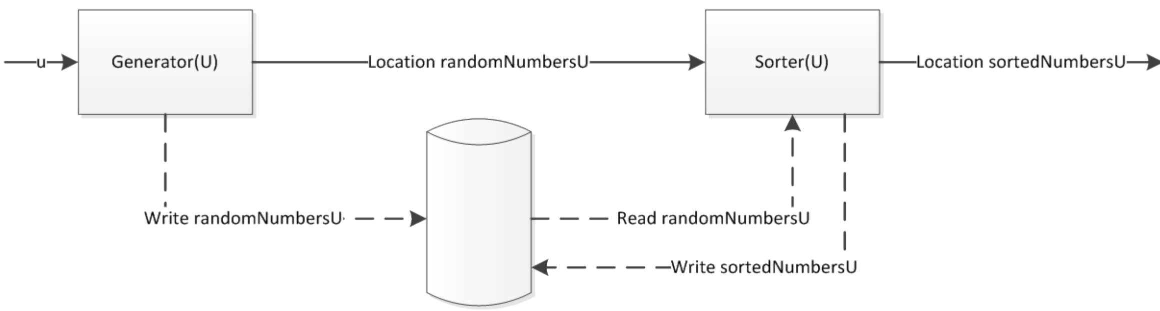

Figure 7: Communication between workflow components

Storage of files is needed for the communication of the simple workflow application. To make sure that all components have access to the workflow files, the framework needs a file system that allows the entities to access these files. There are several possibilities to gain file access. The current *workflow framework* makes use of a shared disk. This shared disk is accessible for all nodes in the framework. This is easy to implement, but a disadvantage of this implementation is that multiple applications may have concurrent access to the same file. A possible way to solve this problem is by using unique filenames for every application. An Orchestrator may use the name of the file in combination with its own unique ID.

Another possibility for file communication is file transport to the right locations. In that case, the FAO has the responsibility to copy files from the local repository of one worker node to the local repository of another worker node. Therefore the framework needs to know the file dependencies of a component of the workflow application. The current framework is not aware of these implementation details, and because of the additional adaptations to the framework this solution is not implemented.

A final solution for file communication could be a FLUENT File Manager that is responsible for all data storage and data access.

6.2.2 Sorter function

The randomly generated numbers of sets U and X_i of the simple workflow application are sorted in ascending order. Because these sets are sorted, the intersection of the two sets can be calculated much easier. By using the sorter function, there are also additional tasks created that generate additional workload.

The sorter function uses insertion sort for its implementation. Insertion sort is a simple sorting algorithm that builds the final sorted list one item at a time. Insertion sort is much less efficient than more advanced sorting algorithms. However, it has some advantages. Insertion sort is an in-place algorithm, i.e., it only requires a constant amount of additional memory space, and insertion sort is easy to implement. The running time of insertion sort is $O(n^2)$. Sorting large sets will take some time, and that creates workload for the workflow application.

6.3 Workflow application deployment

The framework entities of FLUENT that are responsible for application deployment in the *streaming framework* are the FGT and the FAO. FGT entities are the clients of the framework. To deploy an application in FLUENT, a framework client has to create a new application description. A FLUENT application description defines all used components together with the bindings between them and the deployment information. To create an application description, FLUENT needs a library with the description and implementation of the components types of that application. Figure 3 shows the activities in the application life cycle.

When the user wants to submit a new application, the FGT sends a request to the FAO to create a new application description. The FGT can be used to compose the application and to create the bindings between the components of the application. The framework installs and instantiates the components. Installation makes sure that the component types are available at the right locations, and instantiation creates the processes, i.e., the actual components. For all configuration settings that are made to the application, the FGT sends a configuration request to the FAO while the FAO handles that request. The FGT is only used for visualization and user configuration, while the FAO manages the actual application. After the application is configured, and all bindings between the components are set, the user can create a deployment configuration. The final step to actually deploy the application and activate the components is performed by the user clicking the *deploy application* button in the FGT. The FGT will send a request to the FAO to deploy the application, and the FAO will handle the actual deployment. Figure 8 shows a sequence diagram of (a part of) the described scenario of application deployment in the FLUENT *streaming framework*.

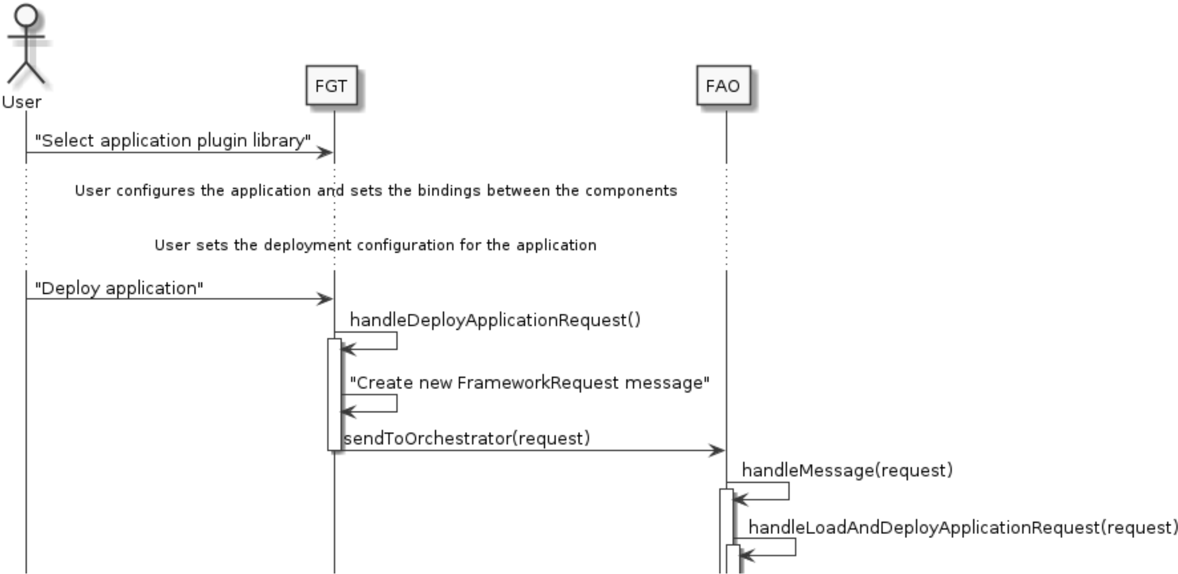

Figure 8: Sequence diagram of application deployment in the streaming framework

In case of a workflow application, there is no user that creates a deployment configuration. Therefore there is only a FAO needed that manages the application and no FGT entity. To deploy the workflow application, the FAO must be able to create a deployment configuration. In workflow terms, the FAO must be able to schedule the workflow tasks over the available resources.

To schedule the workflow tasks, the FAO needs an application description that has at least the information about the bindings between the components. The bindings between the components are the data dependencies in the workflow. Using this information, the FAO knows which tasks depend on each other, and which tasks need to be finished before a depending task can start. The FAO is not able to decide which interfaces between the components have to be connected. This can only be done by the user who has developed the workflow. Because FLUENT already has an application description for describing a component-based (workflow) application, this application description with information about the components is used, such that the FAO knows all tasks that are part of the workflow. The user has to create the application description with information about the dependencies between the tasks. The workflow language that is described in Section 4.3 can be used for describing a workflow. To use the workflow language, FLUENT must be able to translate it into an application description. This functionality is currently not present in the framework.

Besides the components of the workflow, the FAO needs a list of available resources. This list of available resources is obtained by sending a request to the FRM. The communication with the FRM and the actual workflow scheduling is described in the next sections.

6.4 FLUENT Resource Manager

Resource discovery and resource management is important for the *workflow framework*, therefore the FLUENT Resource Manager is an important entity. The FRM is specified in the original FLUENT design document of SAN.

6.4.1 UDP connection with other entities

The framework entities communicate with each other through the User Datagram Protocol (UDP) [10]. UDP is a communications protocol that offers a limited amount of service when messages are exchanged between computers in a network that makes use of the Internet Protocol (IP). UDP is an alternative to the Transmission Control Protocol (TCP). Like TCP, UDP uses the Internet Protocol to actually get a data unit (datagram) from one computer to another. Unlike TCP, UDP does not provide sequencing of the packets that the data arrives in. This means that an application that uses UDP and that use messages of multiple packets does not know whether the entire message has arrived and is in the right order. Network applications, like the FLUENT entities, that want to save processing time and do not care about the receiving of the message may prefer UDP instead of TCP.

To setup a UDP connection between two framework entities, one entity has to behave like an UDP server and the other entity has to behave like an UDP client. The FRM will be configured as an UDP server and the Device Managers and Orchestrators are the UDP clients that are connected to the FRM.

6.4.2 Resource capacities

The FRM has to communicate with the Device Managers that are connected to the framework. The FRM and the connected Device Managers exchange information like the resource capacities of the FDMs. The available Device Managers in the framework are obtained when the FDM entities connect to the FRM UDP server at their initialization phase. The FRM will send every period a request to all connected Device Managers to get their CPU and RAM capacities. Figure 9 shows a sequence diagram of the CPU capacity request that is sent from the FRM to a FDM. When the message is received at the FDM side, the FDM asks for its local capacities and sends a framework reply message back to the FRM. The FRM adds the received capacities to its Capacity table. The Capacity table is considered in Section 5.2.

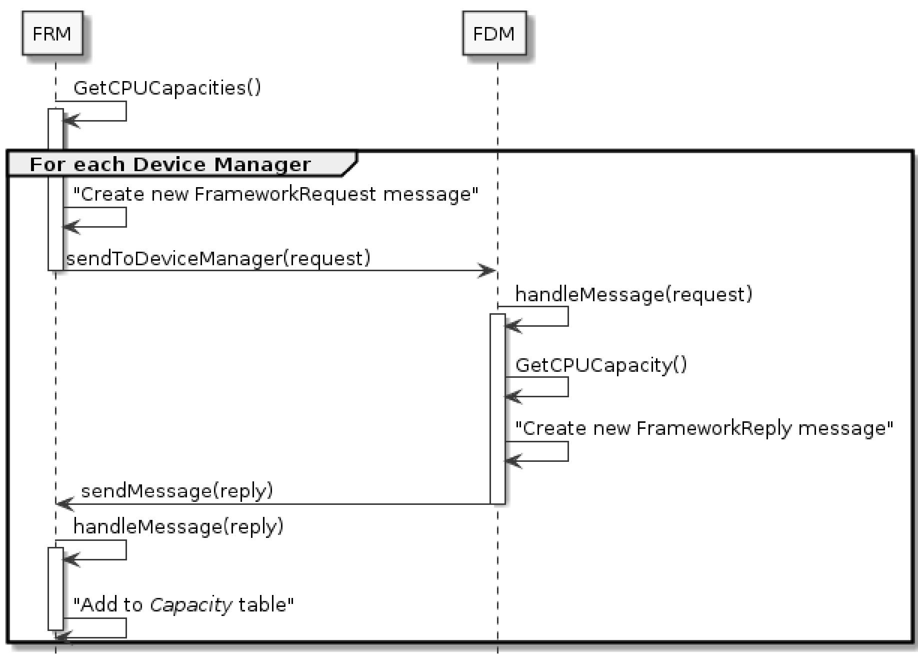

Figure 9: Sequence diagram of CPU capacity request between FRM and FDM

For testing purposes and for good visualization of the scheduling of the workflow tasks, the Device Managers have a fixed CPU and RAM capacity. This is a number between 0 and 100 that indicates the percentage of free capacity of the node. The number is configurable by a user and is given to the Device Manager at its initialization phase.

6.4.3 Reservation of resource capacities

The FRM has to communicate with the Orchestrators in the framework. The Orchestrator is the most important part of the *workflow framework*. The FAO is responsible for scheduling the workflow. Section 6.5 describes the implementation of the workflow scheduler. To do the scheduling, the FAO has to know the resources that are available in the framework and the capacities of the nodes. The FAO uses the connection with the FRM to send requests to get the node capacities, requests to reserve node capacities, and requests to release reserved node capacities. When the FAO sends a resource reservation request or a resource reservation release request to the FRM, the FRM knows that it has to respectively decrease or increase the capacity of a certain resource, if the request is accepted by the FRM.

6.5 Workflow scheduling

At this point FLUENT is able to execute a workflow. The Orchestrator is able to deploy an application, the Resource Manager is aware of the resources and their capacities, and there is an implementation of a workflow application. The last important part of the *workflow framework* is the ability to schedule the individual tasks of a workflow over the available resources.

6.5.1 Scheduling policies

The workflow scheduler of the Orchestrator can handle three different scheduling policies, i.e., round-robin scheduling, largest free resource first, and smallest free resource first. This section will explain these scheduling policies by an example of the scheduling.

Suppose there are five workflow tasks (T1, T2, T3, T4, T5) and three computing resources (R1, R2, R3). All tasks need 10% free capacity to execute. Resource R1 has 50% free capacity available, resource R2 has 30% free capacity available and resource R3 has 15% free capacity available.

Round-robin scheduling

Round-robin scheduling is an algorithm that assigns the resources over the workflow tasks in circular order. Round-robin is simple, easy to implement, and starvation free. Starvation free means that all workflow tasks will always receive an amount of the CPU time of a resource.

Next example explains the round-robin scheduling of the FAO. The example makes use of the five workflow tasks and the three computing resources that are described before. The five tasks are scheduled starting with task T1 and ending with task T5. Figure 10 shows an example of round-robin scheduling. The resources are assigned to the tasks in circular order, starting with R1 assigned to T1, R2 assigned to T2, R3 assigned to T3, R1 assigned to T4, and finally R2 assigned to T5.

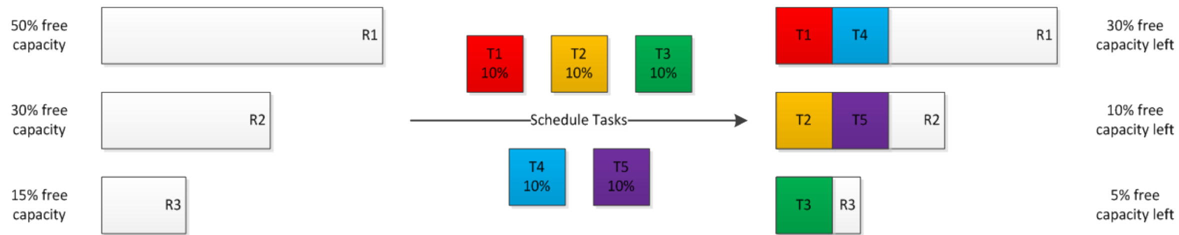

Figure 10: Example of round-robin scheduling

Largest free resource first

The scheduler that uses the largest free resource first policy assigns the resource with the largest free capacity to a task. When multiple resources have equal free capacity, the scheduler uses the first resource of them that is in the resource list.

Next example explains the largest free resource first scheduling policy. This example makes use of the same tasks and resources used in the example in Figure 10. Figure 11 shows an example of the scheduler that uses largest free resource first. Resource R1 has 50% free capacity which is largest, so T1 is scheduled on R1. For the next task, R1 still has the largest free capacity available, such that T2 is also scheduled on R1. For task T3 there are two resources with equal free capacity, R1 and R2. R1 comes first in the resource list, so T3 is scheduled on R1. Now, R2 has the largest free capacity, so T4 is scheduled on R2, and finally, T5 is scheduled on R1.

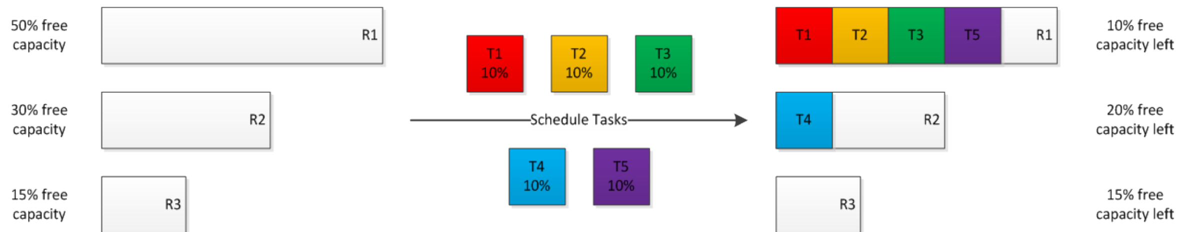

Figure 11: Example of largest free resource first scheduling

Smallest free resource first

The third scheduling policy that is available for scheduling workflows is smallest free resource first. Using smallest free resource first, the resource utilization is maximized. When multiple resources have equal free capacity, the scheduler uses the first resource that is in the resource list.

The example in Figure 12 shows an example of the smallest free resource first scheduling policy. Resources R3 has 15% free capacity which is smallest, so T1 is scheduled on R3. R3 has 5% free capacity left, which is too small to schedule a task. Next tasks are scheduled on the resource that has enough free capacity available, so T2, T3 and T4 are scheduled on resource R2. The last task, T5, is scheduled on R1.

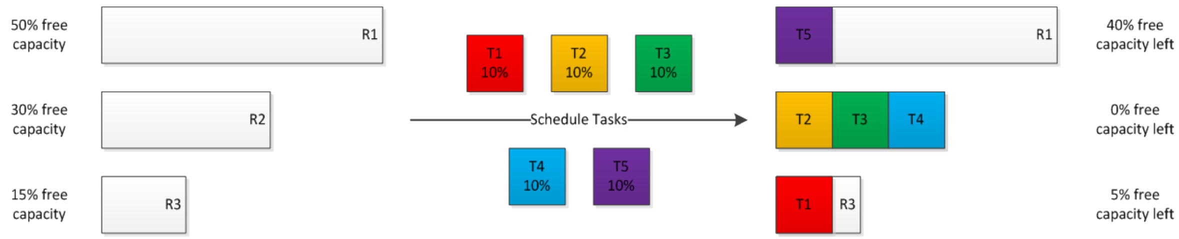

Figure 12: Example of smallest free resource first scheduling

6.5.2 Scheduling workflow tasks

The scheduling of a workflow is shown in Figure 13. This figure gives an example of the scheduling of a workflow using round robin scheduling.

Before the FAO can schedule the tasks of the workflow, it first has to fetch the application description file from the FGR. The application description describes all components and bindings between the components of a FLUENT application. The FAO has an application state machine that helps with managing the application. After the application state machine has loaded the application description, the FAO can obtain a list of all workflow tasks, which are the components of the application. Using the list of components and the Worker nodes table, the FAO can start with scheduling the workflow.

The actual scheduling, that is done for each component in the list, starts with defining the node ID and Dock ID for the component. According to the settings for Dock usage where the scheduler can use a single Dock per node or multiple Docks per node, the scheduler returns a Dock ID. According to

the scheduling policy that is used, the scheduler returns a node ID. The scheduler updates the capacities of the resources in the local Worker nodes table. After the node ID and the Dock ID are defined, the FAO sends a request to the FRM to reserve the resource capacity that is needed for the component. When the reservation request is accepted by the FRM, the FRM has to send an acknowledgement back to the FAO. If the reservation request is not accepted, the FAO has to reschedule the task. When the resource capacity is successfully reserved, the FAO sets the Dock ID and the node ID of the component of the application description of the application state machine. It also adds an entry for the component in its local Workflow components table. The FAO uses this table to manage the resource reservations for all tasks of a workflow. When a task of a workflow is completed, the FAO knows the amount of reserved capacity that has to be released.

Finally, when all the tasks of the workflow are scheduled, the FAO sets the updated component instances of the application state machine, and it saves the application description.

After scheduling the workflow the FAO sends a request to the Device Manager(s) to load and deploy the application, after which the FDM entities create their Docks that have to execute the workflow tasks. All tasks of the workflow are scheduled offline and all components are activated and initialized at once. Another possibility is to have a more dynamic scheduling behavior, such as scheduling the components at runtime. In that case a component is scheduled at the moment that all components where it depends on are finished and all input files are available. Dynamic scheduling requires late component binding, while static scheduling requires that all components have to be bonded when an application is activated. As mentioned in Section 6.2.1 the framework has to be aware of the implementation details of the components of the workflow application. Because of the additional adaptations to the framework that are needed, and therefore time, only the offline scheduling is implemented in the framework.

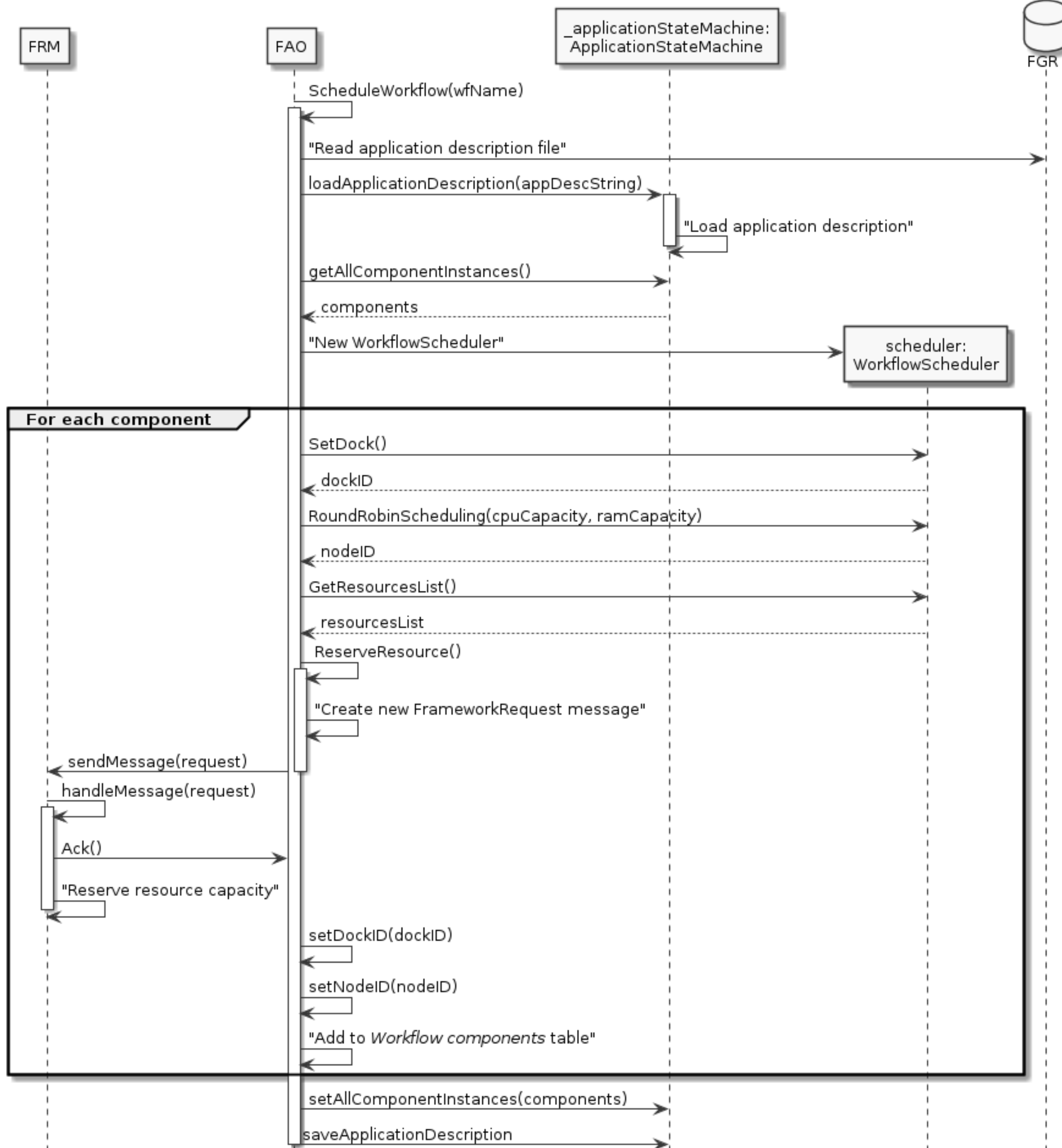

Figure 13: Sequence diagram of workflow scheduling using round robin scheduling

6.5.3 Workflow task finalization

The scheduler on the FAO schedules the workflow tasks over the available resources. When a task is scheduled, that task needs some capacity of a certain resource. Therefore the FAO sends a request to the FRM to reserve that capacity of the resource. When a task of the workflow is finished with its execution, the reserved capacity of its resource has to be released. Therefore the FAO has to send a request to the FRM to release the reserved capacity. The FLUENT *streaming framework* is developed for applications that run indefinitely, and it does not know whether a component of an application is finished with its execution or not.

To release the reserved resource capacities, the definition and implementation of a component instance needs some adaptations. First of all, the framework has to know whether a component is finished or not. To do this, every component sends a notification to the Orchestrator when it is finished with its execution. When the FAO receives this notification message, it will send a request to the FRM to release the reserved capacity of the resource. When all components have sent a notification to the FAO that their execution is finished, the FAO will request its application state machine to close the application. The finalization of a workflow execution is shown in Figure 14.

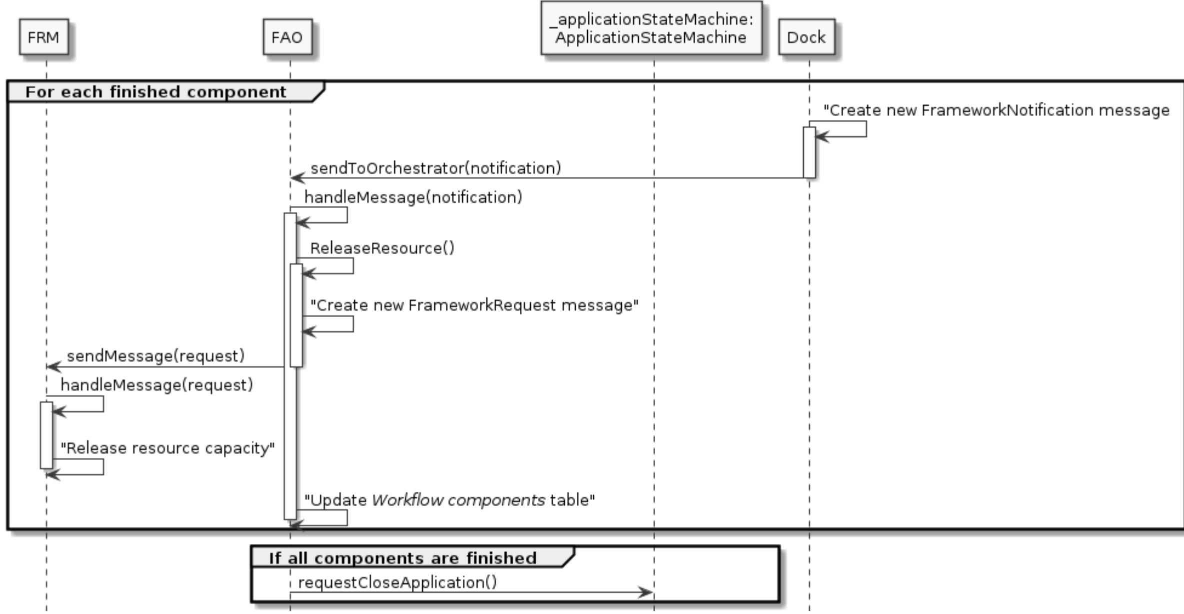

Figure 14: Sequence diagram of workflow finalization

Chapter 7

Performance evaluation

We want to know what the effect is of the simple workflow application on the FLUENT framework. Except that we want to know whether the workflow application will be executed correctly by the framework, we also want to know what the effect is using different configurations by measuring the execution time. This section first describes the experimental setup and the experiments, and finally the obtained results are analyzed.

7.1 Experimental setup

Using the simple workflow application for this experiment, there are some parameters that can be varied and that give a high degree of freedom to the experiment. The parameters that can be varied are:

- The size of the set U ($2^{15} \leq 2^{u} \leq 2^{19}$)
- The size of the sets X_i ($2^{10} \leq 2^{x} \leq 2^{14}$)
- The number of sets X_i ($1 \leq i \leq n$), which defines the number and type of components
- The number of Docks per Device Manager
- The number of Device Managers (virtual machines) per physical machine
- The capacities of the worker nodes (virtual machines)

We want to setup two different experiments to test the *workflow framework*, using the simple workflow application described in Section 4.4 that creates the workload. We want to measure the execution time of the workflow application using fixed application parameters and using different configurations of FLUENT, and we want to test the workflow scheduler using a fixed configuration for the FLUENT framework and workflow application, and using different capacities of the nodes that are used for the experiment.

The experiments are performed on a small computer network that consists of three computers. The computers are connected to each other via a switch and can communicate with each other over the network. All computers are running on Windows 7 OS, they are running Oracle VirtualBox and they contain one or two virtual machines. Every virtual machine has read and write access to the shared disk that is located on LAPTOP-2. Figure 15 gives an overview of the computer network that is used for the experiment and Table 1 shows the specification of the computer nodes that are used.

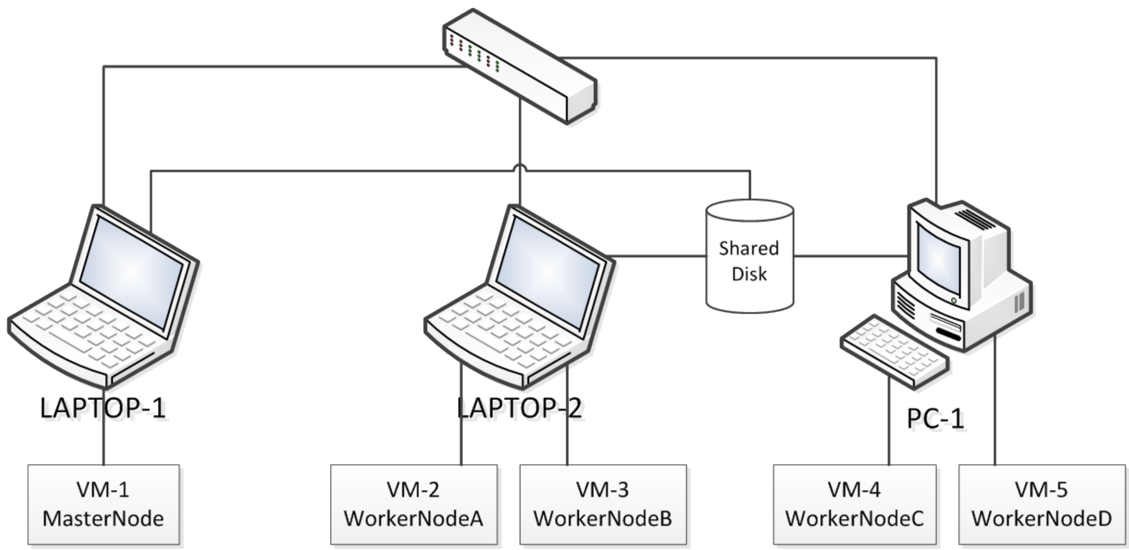

Figure 15: Experimental computer network

Computer name	Processor	Memory	OS
LAPTOP-1	Intel Core i7-740QM, 1.73 GHz	4 GB	Windows 7 Ultimate
LAPTOP-2	Intel Core i7-2630QM, 2.00 GHz	4 GB	Windows 7 Enterprise
PC-1	Intel Core 2 Quad Q6600, 2.40 GHz	3 GB	Windows 7 Enterprise

Table 1: Specification of the computer nodes

The configuration of the virtual machines that are installed on the computers is shown in Table 2. All virtual machines are running on Ubuntu 12.04 OS and are using one CPU core and 1024 MB RAM of the host machine. The FLUENT framework entities that are running on the master node are the FFM, the FRM and a FAO. The other four virtual machines all have one single FDM deployed on it. These are the worker nodes of the framework.

Virtual machine	Physical machine	Memory	Number of CPUs	OS	Framework entities
MasterNode	LAPTOP-1	1024 MB	1	Ubuntu 12.04	FFM, FRM, FAO
WorkerNodeA	LAPTOP-2	1024 MB	1	Ubuntu 12.04	FDM
WorkerNodeB	LAPTOP-2	1024 MB	1	Ubuntu 12.04	FDM
WorkerNodeC	PC-1	1024 MB	1	Ubuntu 12.04	FDM
WorkerNodeD	PC-1	1024 MB	1	Ubuntu 12.04	FDM

Table 2: Configuration of the virtual machines

7.2 Experiments

We want to setup two different experiments to test the FLUENT *workflow framework*. We want to measure the execution time of the simple workflow application, and we want to test the workflow scheduler of the FAO.

7.2.1 Execution time of simple workflow application

To measure the execution time of the simple workflow application there are some parameters that can be varied and that will increase the testing domain. To make the testing domain smaller, we want to use the simple workflow application with fixed parameters. It is obvious that an application that uses bigger sets U and X_i have a longer execution time because especially the sort and the intersection functions take much more time.

The following fixed parameters are used for this experiment:

- The size of set U is set to 2^{19}, $u = 19$
- The size of sets X_i is set to 2^{14}, $x = 14$
- The number of sets X_i is set to 2, $i = 2$, the number of tasks is 11
- The available capacities of the used nodes are set to 100%
- The round-robin scheduler is used

The tests that are performed for this experiment:

1a. One worker node that uses a single Dock
1b. One worker node that uses multiple Docks
2a. Two worker nodes that use a single Dock
2b. Two worker nodes that use multiple Docks
3a. Three worker nodes that use a single Dock
3b. Three worker nodes that use multiple Docks
4a. Four worker nodes that use a single Dock
4b. Four worker nodes that use multiple Docks

Performing these experiments and using this configuration, we want to test the differences in execution time, where the number of worker nodes is increased, and the number of Dock entities per FDM is varied. Further we can test with this experiment the differences in execution time between intra-Dock and inter-Dock communication.

7.2.2 Workflow scheduler

With the second experiment we want to test the workflow scheduler that is used by the FAO. All three scheduling policies will be tested by using a different configuration of the capacities of the worker nodes and by varying the number of worker nodes.

The following fixed parameters are used for this experiment:

- The size of set U is set to 2^{19}, $u = 19$
- The size of sets X_i is set to 2^{14}, $x = 14$
- The number of sets X_i is set to 2, $i = 2$, the number of tasks is 11
- There are multiple Docks per worker node
- The required capacity per workflow task is set to 5%

The tests that are performed for this experiment:

1a. Round-robin, one worker node, 55% free capacity
1b. Round-robin, two worker nodes, respectively 40% and 25% free capacity
1c. Round-robin, three worker nodes, respectively 30%, 20% and 15% free capacity
1d. Round-robin, four worker nodes, respectively 30%, 10%, 15% and 30% free capacity
2a. Largest free resource first, one worker node, 55% free capacity
2b. Largest free resource first, two worker nodes, respectively 40% and 25% free capacity
2c. Largest free resource first, three worker nodes, respectively 30%, 20% and 15% free capacity
2d. Largest free resource first, four worker nodes, respectively 30%, 10%, 15% and 30% free capacity
3a. Smallest free resource first, one worker node, 55% free capacity
3b. Smallest free resource first, two worker nodes, respectively 40% and 25% free capacity
3c. Smallest free resource first, three worker nodes, respectively 30%, 20% and 15% free capacity
3d. Smallest free resource first, four worker nodes, respectively 30%, 10%, 15% and 30% free capacity

Performing these experiments and using this configuration, we want to test the workflow scheduler that is used by the FAO. We want to test whether it schedules the tasks of the simple workflow over the available nodes as expected.

7.3 Experimental results

The results from the first experiment in which we want to measure the execution time of the simple workflow application, show that there are no big differences in execution time when the number of worker nodes is increased. The results of this experiments show that experiments 1b, 2b, 3b, and 4b take more time than their corresponding experiments 1a, 2a, 3a, and 4a. This means that intra-Dock communication takes more time than inter-Dock communication. When using multiple Docks there is additional time caused by the communication from one Dock to another. The results of tests *a* that use a single Dock per node are shown in the graphs of Figure 16 and Figure 17. Figure 16 shows the execution times of tasks *UsIn*, *GenU*, *GenX1*, *GenX2*, *SorX1* and *SorX2*, while Figure 17 shows the execution times of tasks *SorU*, *Demux*, *IntY1*, *IntY2* and *Avg*. The results of tests *b* that use multiple Docks per node are shown in the graphs of Figure 18 and Figure 19, where Figure 18 shows the execution times of tasks *UsIn*, *GenU*, *GenX1*, *GenX2*, *SorX1*, and *SorX2*, and Figure 19 shows the execution times of tasks *SorU*, *Demux*, *IntY1*, *IntY2*, and *Avg*.

The results of the first experiment do not show a linear decrease in execution time when the number of worker nodes is increased. However, there is a small decrease in execution time, especially in tests *b* of the experiment. In Table 3 and Table 4 in Appendix A the execution times are shown. The execution times in these tables show that the task *SorU* is in all cases the bottleneck in the workflow. By dividing the workload better over the workflow tasks, the bottleneck should be removed, and a more linear decrease in execution time is expected when the number of worker nodes is increased.

Therefore this experiment is performed using the following adapted fixed parameters:

- The size of set U is set to 2^{16}, $u = 16$
- The size of sets X_i is set to 2^{14}, $x = 14$
- The number of sets X_i is set to 4, $i = 4$, the number of tasks is 19
- There are multiple Docks per worker node
- The capacities of the used nodes are set to 100%
- The round-robin scheduler is used

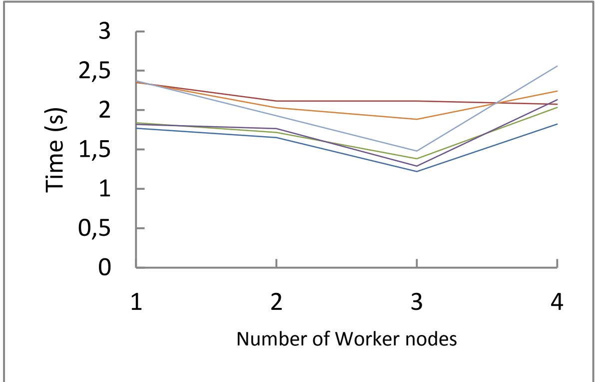

Figure 16: Graph with results of tests a of experiment 1

Figure 17: Graph with results of tests a of experiment 1

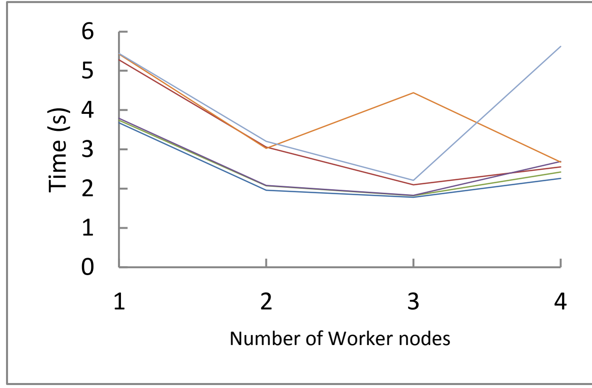

Figure 18: Graph with results of tests b of experiment 1

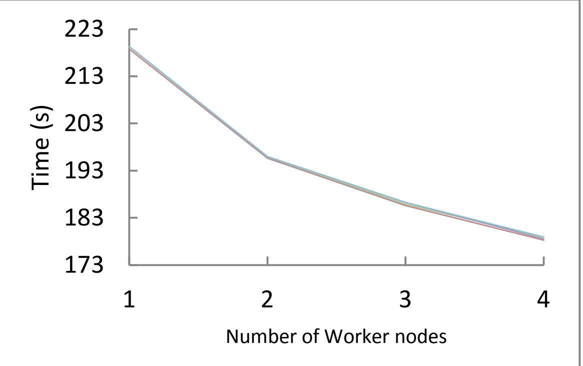

Figure 19: Graph with results of tests b of experiment 1

The results of the experiment using the adapted fixed parameters show a more decreasing execution time compared to the results of the previous test. However, the decrease is not linear and there are some peaks in the graph. The graph with the results is shown in Figure 20. The deviations can be caused by the fact that there is a delay caused by the network. Further, the scheduler divides the tasks over the resources without looking at the tasks themselves, such as scheduling all *Generator* tasks on different nodes and scheduling the $Generator(U)$ and $Sorter(U)$ tasks, and the $Generator(X_n)$ and $Sorter(X_n)$ tasks on the same nodes. Table 5 in Appendix A shows the execution times.

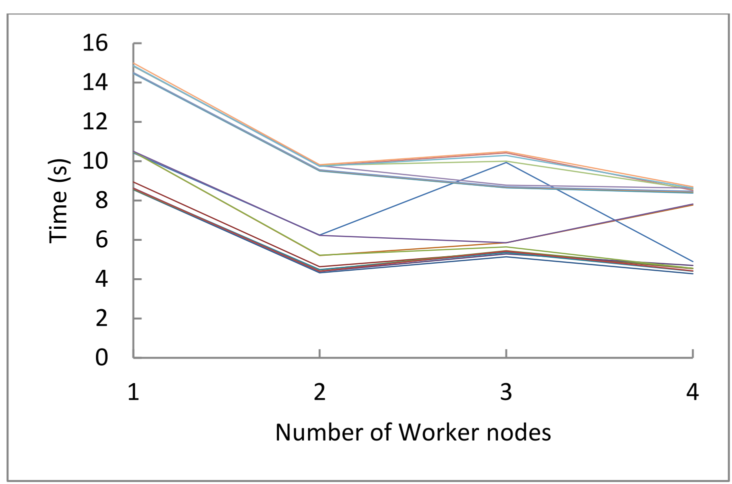

Figure 20: Graph with results of experiment 1 using adapted parameters

As the result of the second experiment shows, in which the performance of the workflow scheduler is tested, the scheduler on the FAO works as expected. The second experiment is performed using respectively one, two, three, and four worker nodes with different capacities and it is performed for the round-robin, largest free resource first and smallest free resource first scheduling policies. Figure 21 shows the expected scheduling output for round-robin, largest free resource first, and smallest free resource first, using one worker node with a free capacity of 55%. The result of this experiment shows that the workflow is scheduled as expected. Figures 26, 30, and 34 in Appendix A show the screenshots of tests *a* of experiment 2.

UsIn	GenU	GenX1	GenX2	SorU	SorX1	SorX2	IntY1	IntY2	Demux	Avg	0% free capacity left

Figure 21: Expected scheduling output using one worker node

Figure 22 shows the expected output of respectively test 1b, test 2b, and test 3b of experiment 2. Figures 27, 31, and 35 in Appendix A show the screenshots of the results of these tests. The screenshots indicate that the scheduler works as expected.

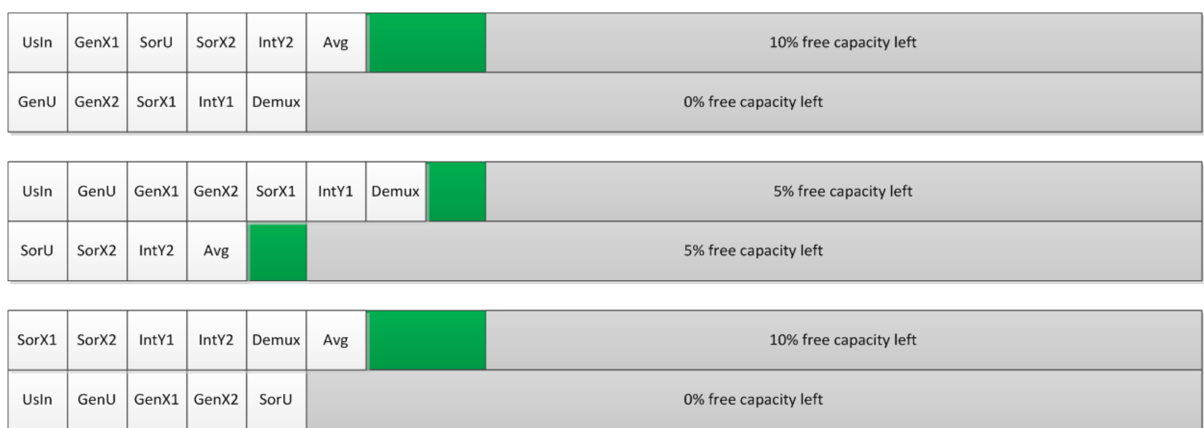

Figure 22: Expected scheduling output using two worker nodes

Figure 23 shows the expected output of respectively test 1c, test 2c, and test 3c of experiment 2. Figures 28, 32, and 36 in Appendix A show the screenshots of the results of these tests. The screenshots indicate that the scheduler works as expected.

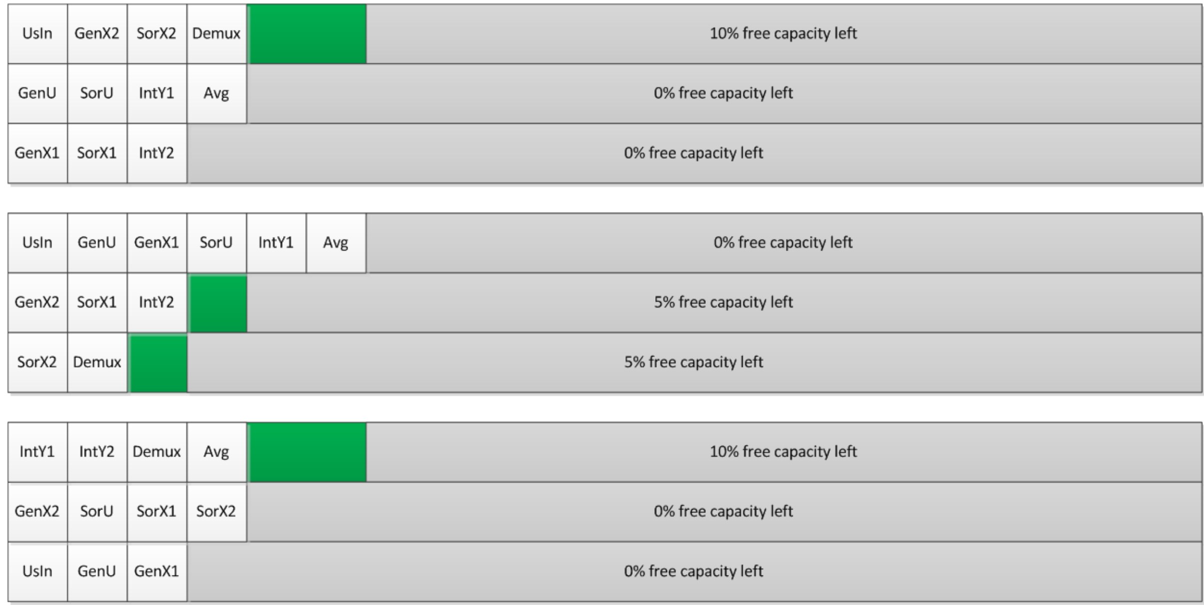

Figure 23: Expected output using three worker nodes

Figure 24 shows the expected output of tests 1d, 2d, and 3d of experiment 2. The screenshots on Figures 29, 33, and 37 in Appendix A show the results of these tests. The screenshots indicate that the scheduler works as expected.

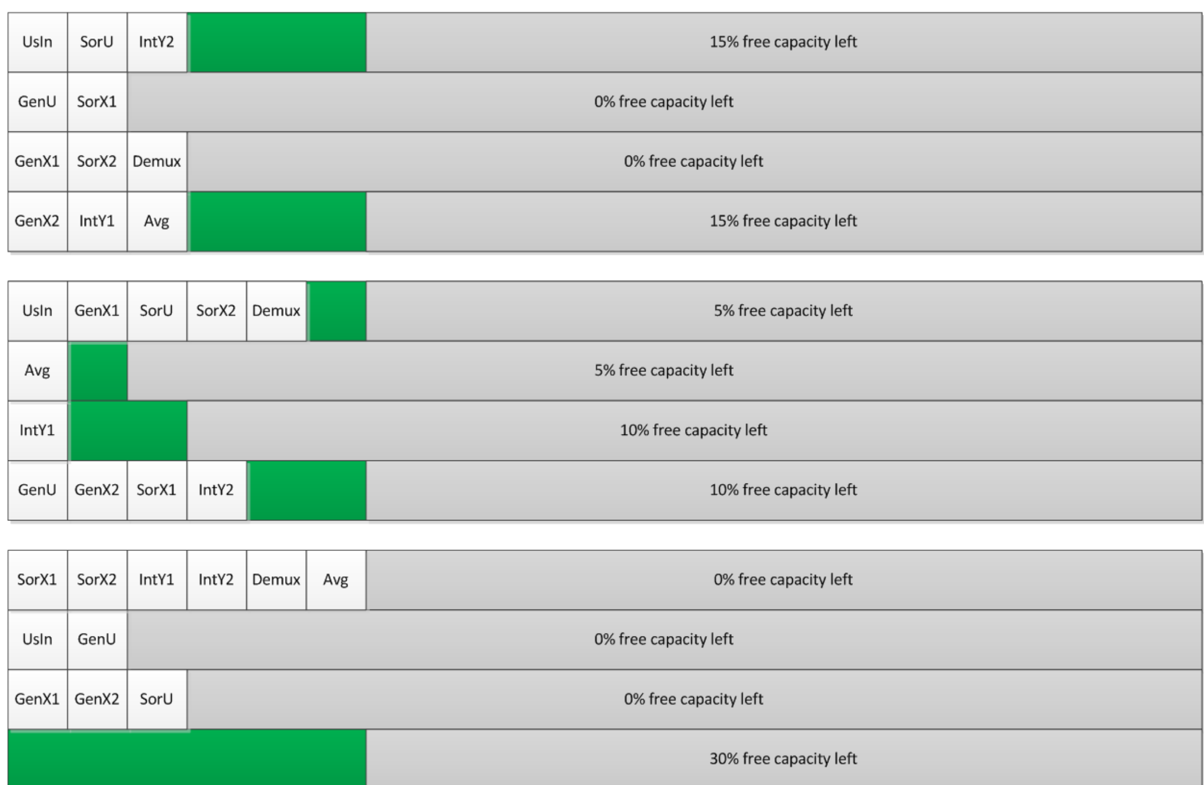

Figure 24: Expected scheduling output using four worker nodes

Another experiment for testing the scheduler can be performed by executing two applications at a time in the framework. Therefore, there are two FAO entities that both manage a workflow application with the following parameters. The workflow has 7 tasks (the number of sets X_i is 1, $i = 1$), there are multiple Docks per worker node, the required capacity per task is 5%, and the largest free resource scheduling policy is used. There are four worker nodes which respectively have capacities 25%, 10%, 15%, and 20%. The first Orchestrator starts scheduling at time t_1 and the second starts scheduling at time t_2, where $t_1 < t_2$.

The FAO uses offline scheduling, so the first FAO schedules first its tasks, after which the second FAO schedules its tasks. Figure 25 shows the expected scheduling output. The tasks that are colored red are the tasks of the first FAO, while the tasks that are colored blue are the tasks of the second FAO.

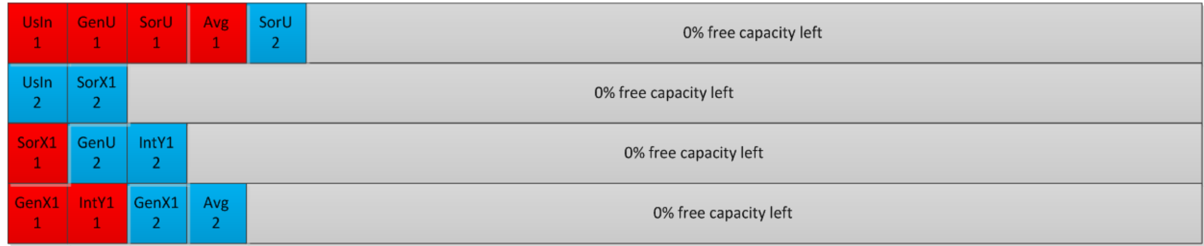

Figure 25: Expected scheduling output using two applications

The results of the test are shown in the screenshots in Figure 38 and Figure 39 in Appendix A. These screenshots indicate that the scheduler works as expected.

Chapter 8

Conclusion and evaluation

This chapter concludes the work that is performed for this Master project and that is described in this thesis. Furthermore, the suitability of FLUENT for workflow management is evaluated.

FLUENT is originally being designed and developed by SAN as a framework for video processing. The intention of FLUENT is to perform life cycle management for arbitrary service-oriented applications. Because of the resource management capabilities of FLUENT it seems natural to investigate whether FLUENT can be used as a workflow management system.

Before FLUENT can be used as a workflow management system, the important characteristics of a workflow framework need to be identified. Therefore the FLUENT framework is reviewed on its responsibilities in the *streaming framework*, and the responsibilities and requirements that are needed for workflow support are identified. One of the most important parts of a workflow framework is the workflow engine. The workflow engine is responsible for managing and executing the workflow. FLUENT already contains a framework entity that is responsible for application management and an entity that is responsible for the execution of an application. These entities are respectively the FAO and the FDM. These framework entities are the important part of the *workflow framework*. The FAO is extended with functionalities to schedule the tasks of a workflow and the Docks that are managed by the FDM execute the individual workflow tasks. Further, a simple synthetic workflow application is developed to test all implementations of the *workflow framework* and the other adaptations to FLUENT.

The last part of this thesis describes the experiments that are performed to test the functionalities that are implemented for FLUENT to support workflows. There are two different experiments that are performed. One to measure the execution time of the simple workflow application, and one experiment to test the workflow scheduler on the FAO. The results from the first experiment, in which the execution time is measured, show that there are no big differences in execution time for the simple workflow application when the number of worker nodes is increased. The execution time is not decreasing linearly when the number of nodes is increased linearly. The workload in the workflow is not balanced, there is one dominating task. When creating a better balanced workload, the execution time will decrease, but not linear. The results of the second experiment, in which the workflow scheduler is tested, show that scheduling of a workflow works as expected.

From this all we can conclude that FLUENT is suitable for workflow management. For this project there are some extensions and adaptations made to enable the support for workflows. As future work there are more extensions and adaptations possible to improve the *workflow framework*. The scheduling of the workflow can have a more dynamic behavior, such as scheduling the components at runtime. In that case a workflow task is scheduled if all tasks where it depends on are finished and all input files are available. Further, the scheduling can be improved by considering multiple types of

resources. For better visualization, the current implementation makes use of an abstraction of CPU and RAM capacities. This can be improved by using real CPU and RAM constraints. Also the file system that is responsible for the communication between the workflow tasks can be improved. The current workflow framework makes use of a shared disk, but by using a more advanced File Manager the problems that come with the use of a shared disk can be avoided.

References

[1] R.H. Mak: "Resource-aware Life Cycle Models for Science-oriented Applications managed by a Component Framework", Computer Science Reports 13-7, 2013

[2] I. David, B. Orlic, R.H. Mak, J. Lukkien: "Towards Resource-Aware Runtime Reconfigurable Component-Based Systems", IEEE 6th World Congress on Services, 2010

[3] B. Orlic, I. David, R.H. Mak, J. Lukkien: "Dynamically Reconfigurable Resource-Aware Component Framework: Architecture and Concepts", Proceedings of the 5th European Conference on Software Architecture, 2011

[4] "ViCoMo", http://www.vicomo.org

[5] J.W. Verkooijen: "FLUENT as a Workflow Framework: a Pilot Study", Report of Pre-study Master Project, 2013

[6] R. Mak, B. Orlic, I. David, A. Kuzmanovska: "FLUENT Design Document", 2012

[7] "Extensible Markup Language (XML)", http://www.w3.org/XML/

[8] "Introducing JSON", http://json.org/

[9] "Poco C++ Libraries", http://pocoproject.org/

[10] "RFC 768 - User Datagram Protocol", http://tools.ietf.org/html/rfc768

Appendix

A: Experimental results

The results of the experiments that are described in Chapter 7 are shown in this section. Next tables and screenshots show the output of different experiments. Table 3 shows the time results of the workflow tasks in seconds using a single Dock, and Table 4 shows the time results of the workflow tasks in seconds using multiple Docks.

Number of nodes: 1 Single Dock		**Number of nodes: 2 Single Dock**		**Number of nodes: 3 Single Dock**		**Number of nodes: 4 Single Dock**	
Component	**Time (s)**	**Component**	**Time (s)**	**Component**	**Time (s)**	**Component**	**Time (s)**
UsIn	1.77	UsIn	1.65	UsIn	1.22	UsIn	1.82
GenU	2.35	GenU	2.11	GenU	2.12	GenU	2.07
GenX1	1.84	GenX1	1.72	GenX1	1.38	GenX1	2.03
GenX2	1.82	GenX2	1.77	GenX2	1.29	GenX2	2.13
SorU	170.87	SorU	170.69	SorU	171.62	SorU	169.03
SorX1	2.36	SorX1	2.03	SorX1	1.89	SorX1	2.24
SorX2	2.37	SorX2	1.93	SorX2	1.48	SorX2	2.56
Demux	170.88	Demux	170.71	Demux	171.63	Demux	169.05
IntY1	171.30	IntY1	170.97	IntY1	171.92	IntY1	169.66
IntY2	171.28	IntY2	170.96	IntY2	172.28	IntY2	169.29
Avg	171.31	Avg	170.98	Avg	172.29	Avg	169.68

Table 3: Results of the first experiment that uses a single Dock per worker node

Number of nodes: 1 Multiple Docks		**Number of nodes: 2 Multiple Docks**		**Number of nodes: 3 Multiple Docks**		**Number of nodes: 4 Multiple Docks**	
Component	**Time (s)**	**Component**	**Time (s)**	**Component**	**Time (s)**	**Component**	**Time (s)**
UsIn	3.68	UsIn	1.96	UsIn	1.78	UsIn	2.26
GenU	5.28	GenU	3.06	GenU	2.10	GenU	2.56
GenX1	3.74	GenX1	2.07	GenX1	1.82	GenX1	2.42
GenX2	3.79	GenX2	2.08	GenX2	1.83	GenX2	2.70
SorU	218.77	SorU	195.67	SorU	185.64	SorU	178.26
SorX1	5.43	SorX1	3.03	SorX1	4.44	SorX1	2.68
SorX2	5.45	SorX2	3.21	SorX2	2.21	SorX2	5.62
Demux	218.78	Demux	195.69	Demux	185.65	Demux	178.27
IntY1	219.26	IntY1	195.96	IntY1	185.92	IntY1	178.91
IntY2	219.28	IntY2	195.97	IntY2	186.29	IntY2	178.92
Avg	219.29	Avg	195.98	Avg	186.30	Avg	178.93

Table 4: Results of the first experiment that uses multiple Docks per worker node

Number of nodes: 1 Multiple Docks		Number of nodes: 2 Multiple Docks		Number of nodes: 3 Multiple Docks		Number of nodes: 4 Multiple Docks	
Component	Time (s)	Component	Time (s)	Component	Time (s)	Component	Time (s)
UsIn	8.55	UsIn	4.32	UsIn	5.13	UsIn	4.27
GenU	8.94	GenU	4.63	GenU	5.37	GenU	4.53
GenX1	8.57	GenX1	4.46	GenX1	5.44	GenX1	4.55
GenX2	8.63	GenX2	4.39	GenX2	5.28	GenX2	4.70
GenX3	8.59	GenX3	4.48	GenX3	5.35	GenX3	4.40
GenX4	8.61	GenX4	4.40	GenX4	5.42	GenX4	4.42
SorU	14.44	SorU	9.49	SorU	8.64	SorU	8.36
SorX1	10.46	SorX1	5.20	SorX1	5.84	SorX1	7.77
SorX2	10.42	SorX2	6.24	SorX2	9.93	SorX2	4.89
SorX3	10.47	SorX3	5.21	SorX3	5.64	SorX3	4.54
SorX4	10.49	SorX4	6.22	SorX4	5.85	SorX4	7.82
Demux1	14.47	Demux1	9.51	Demux1	8.64	Demux1	8.39
Demux2	14.49	Demux2	9.53	Demux2	8.67	Demux2	8.43
Demux3	14.50	Demux3	9.55	Demux3	8.70	Demux3	8.46
IntY1	14.83	IntY1	9.75	IntY1	10.41	IntY1	8.51
IntY2	14.82	IntY2	9.78	IntY2	9.99	IntY2	8.58
IntY3	14.84	IntY3	9.76	IntY3	8.77	IntY3	8.62
IntY4	14.86	IntY4	9.74	IntY4	10.28	IntY4	8.63
Avg	14.98	Avg	9.82	Avg	10.48	Avg	8.69

Table 5: Results of the first experiment using adapted parameters

Table 5 shows the execution times of the first experiment that uses adapted fixed parameters to create a better balanced workload.

```
Resource with id: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 has 55% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using roundRobin scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: fcf1f8a8-cad3-11e3-bf28-0800271c8db1 and on Dock: 11
```

Figure 26: Screenshot of test 1a of experiment 2

The screenshot in Figure 26 shows that there is one resource that has a free capacity of 55%. Using the round-robin scheduler all workflow tasks are scheduled on that resource.

```
Resource with id: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 has 40% free CPU capacity
Resource with id: 7144457a-cad5-11e3-a78a-0800271c8db1 has 25% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using roundRobin scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 7144457a-cad5-11e3-a78a-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 7144457a-cad5-11e3-a78a-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 7144457a-cad5-11e3-a78a-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 7144457a-cad5-11e3-a78a-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 7144457a-cad5-11e3-a78a-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 6641a2d0-cad5-11e3-b5c8-0800271c8db1 and on Dock: 11
```

Figure 27: Screenshot of test 1b of experiment 2

The screenshot in Figure 27 shows that there are two resources. The round-robin scheduler schedules the workflow tasks *UsIn*, *GenX1*, *SorU*, *SorX2*, *IntY2*, and *Avg* on the first resource, and tasks *GenU*, *GenX2*, *SorX1*, *IntY1* and *Demux* on the second resource.

```
Resource with id: 778e6d28-cad7-11e3-a332-0800271c8db1 has 30% free CPU capacity
Resource with id: 7d3fda86-cad7-11e3-80c1-0800271c8db1 has 20% free CPU capacity
Resource with id: 83febf0e-cad7-11e3-af82-0800271c8db1 has 15% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using roundRobin scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 778e6d28-cad7-11e3-a332-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 7d3fda86-cad7-11e3-80c1-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 83febf0e-cad7-11e3-af82-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 778e6d28-cad7-11e3-a332-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 7d3fda86-cad7-11e3-80c1-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 83febf0e-cad7-11e3-af82-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 778e6d28-cad7-11e3-a332-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 7d3fda86-cad7-11e3-80c1-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 83febf0e-cad7-11e3-af82-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 778e6d28-cad7-11e3-a332-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 7d3fda86-cad7-11e3-80c1-0800271c8db1 and on Dock: 11
```

Figure 28: Screenshot of test 1c of experiment 2

The screenshot in Figure 28 shows that there are three resources. Using the round-robin scheduler, the workflow tasks *UsIn*, *GenX2*, *SorX2*, and *Demux* are scheduled on the first resource. The tasks

GenU, *SorU*, *IntY1*, and *Avg* are scheduled on the second resource and tasks *GenX1*, *SorX1*, and *IntY2* are scheduled on the third resource.

```
Resource with id: 1bf0aa0c-cad8-11e3-b470-0800271c8db1 has 30% free CPU capacity
Resource with id: 216efaba-cad8-11e3-b597-0800271c8db1 has 10% free CPU capacity
Resource with id: 2750768e-cad8-11e3-b3e0-0800271c8db1 has 15% free CPU capacity
Resource with id: 2e1d1eae-cad8-11e3-a588-0800271c8db1 has 30% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using roundRobin scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 1bf0aa0c-cad8-11e3-b470-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 216efaba-cad8-11e3-b597-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 2750768e-cad8-11e3-b3e0-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 2e1d1eae-cad8-11e3-a588-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 1bf0aa0c-cad8-11e3-b470-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 216efaba-cad8-11e3-b597-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 2750768e-cad8-11e3-b3e0-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 2e1d1eae-cad8-11e3-a588-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 1bf0aa0c-cad8-11e3-b470-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 2750768e-cad8-11e3-b3e0-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 2e1d1eae-cad8-11e3-a588-0800271c8db1 and on Dock: 11
```

Figure 29: Screenshot of test 1d of experiment 2

The screenshot in Figure 29 shows that there are four resources. The workflow tasks *UsIn*, *SorU*, and *IntY2* are scheduled on the first resource. The tasks *GenU* and *SorX1* are scheduled on the second resource, tasks *GenX1*, *SorX2*, and *Demux* are scheduled on the third node, and *GenX2*, *IntY1*, and *Avg* are scheduled on the fourth node. This test has used the round-robin scheduler.

```
Resource with id: 200b0b36-c16a-11e3-9a8c-0800271c8db1 has 55% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using largestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 200b0b36-c16a-11e3-9a8c-0800271c8db1 and on Dock: 11
```

Figure 30: Screenshot of test 2a of experiment 2

The screenshot in Figure 30 shows that there is one resource with a free capacity of 55%. Using the largest free resource first scheduler all workflow tasks are scheduled on that single resource.

```
Resource with id: bb831518-c16a-11e3-8454-0800271c8db1 has 40% free CPU capacity
Resource with id: c2762d2e-c16a-11e3-8eb8-0800271c8db1 has 25% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using largestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: c2762d2e-c16a-11e3-8eb8-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: c2762d2e-c16a-11e3-8eb8-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: c2762d2e-c16a-11e3-8eb8-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: bb831518-c16a-11e3-8454-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: c2762d2e-c16a-11e3-8eb8-0800271c8db1 and on Dock: 11
```

Figure 31: Screenshot of test 2b of experiment 2

The screenshot in Figure 31 shows the largest free resource first scheduler that uses two resources. The scheduler schedules the workflow tasks *UsIn*, *GenU*, *GenX1*, *GenX2*, *SorX1*, *IntY1* and *Demux* on the first resource, and tasks *SorU*, *SorX2*, *IntY2* and *Avg* on the second resource.

```
Resource with id: 0f9649b6-c16d-11e3-aa72-0800271c8db1 has 30% free CPU capacity
Resource with id: 161d79da-c16d-11e3-9b74-0800271c8db1 has 20% free CPU capacity
Resource with id: 1f6b27c6-c16d-11e3-b202-0800271c8db1 has 15% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using largestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 0f9649b6-c16d-11e3-aa72-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 0f9649b6-c16d-11e3-aa72-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 0f9649b6-c16d-11e3-aa72-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 161d79da-c16d-11e3-9b74-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 0f9649b6-c16d-11e3-aa72-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 161d79da-c16d-11e3-9b74-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 1f6b27c6-c16d-11e3-b202-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 0f9649b6-c16d-11e3-aa72-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 161d79da-c16d-11e3-9b74-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 1f6b27c6-c16d-11e3-b202-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 0f9649b6-c16d-11e3-aa72-0800271c8db1 and on Dock: 11
```

Figure 32: Screenshot of test 2c of experiment 2

The screenshot in Figure 32 shows that there are three resources. The largest free resource first scheduler schedules the tasks *UsIn*, *GenU*, *GenX1*, *SorU*, *IntY1* and *Avg* on the first resource. The tasks *GenX2*, *SorX1*, and *IntY2* are scheduled on the second resource and tasks *SorX2*, and *Demux* are scheduled on the third resource.

```
Resource with id: f1b9fb9e-c16d-11e3-99e5-0800271c8db1 has 30% free CPU capacity
Resource with id: fa15ad56-c16d-11e3-b710-0800271c8db1 has 10% free CPU capacity
Resource with id: 00752708-c16e-11e3-a44e-0800271c8db1 has 15% free CPU capacity
Resource with id: 078ad59c-c16e-11e3-a2e7-0800271c8db1 has 30% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using largestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: f1b9fb9e-c16d-11e3-99e5-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 078ad59c-c16e-11e3-a2e7-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: f1b9fb9e-c16d-11e3-99e5-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 078ad59c-c16e-11e3-a2e7-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: f1b9fb9e-c16d-11e3-99e5-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 078ad59c-c16e-11e3-a2e7-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: f1b9fb9e-c16d-11e3-99e5-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 00752708-c16e-11e3-a44e-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 078ad59c-c16e-11e3-a2e7-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: f1b9fb9e-c16d-11e3-99e5-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: fa15ad56-c16d-11e3-b710-0800271c8db1 and on Dock: 11
```

Figure 33: Screenshot of test 2d of experiment 2

The screenshot in Figure 33 shows that there are four resources. The workflow tasks *UsIn*, *GenX1*, *SorU*, *SorX2* and *Demux* are scheduled on the first resource. The task *Avg* is scheduled on the second resource, task *IntY1* is scheduled on the third node, and *GenU*, *GenX2*, *SorX1* and *IntY2* are scheduled on the fourth node. This test has used the largest free resource first scheduler.

```
Resource with id: b3b327cc-cad6-11e3-88a6-0800271c8db1 has 55% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using smallestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: b3b327cc-cad6-11e3-88a6-0800271c8db1 and on Dock: 11
```

Figure 34: Screenshot of test 3a of experiment 2

The screenshot in Figure 34 shows that there is one resource that has a free capacity of 55%. Using the smallest free resource first scheduler all workflow tasks are scheduled on that single resource.

```
Resource with id: 5e8805c4-cad6-11e3-a94c-0800271c8db1 has 40% free CPU capacity
Resource with id: 64c42274-cad6-11e3-a67b-0800271c8db1 has 25% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using smallestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 64c42274-cad6-11e3-a67b-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 64c42274-cad6-11e3-a67b-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 64c42274-cad6-11e3-a67b-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 64c42274-cad6-11e3-a67b-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 64c42274-cad6-11e3-a67b-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 5e8805c4-cad6-11e3-a94c-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 5e8805c4-cad6-11e3-a94c-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 5e8805c4-cad6-11e3-a94c-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 5e8805c4-cad6-11e3-a94c-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 5e8805c4-cad6-11e3-a94c-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 5e8805c4-cad6-11e3-a94c-0800271c8db1 and on Dock: 11
```

Figure 35: Screenshot of test 3b of experiment 2

The screenshot in Figure 35 shows the smallest free resource first scheduler that uses two resources. The scheduler schedules the workflow tasks *SorX1*, *SorX2*, *IntY1*, *IntY2*, *Demux* and *Avg* on the first resource, and tasks *UsIn*, *GenU*, *GenX1*, *GenX2* and *SorU* on the second resource.

```
Resource with id: 2369b540-cad7-11e3-8fae-0800271c8db1 has 30% free CPU capacity
Resource with id: 288fe09e-cad7-11e3-9490-0800271c8db1 has 20% free CPU capacity
Resource with id: 2faaa800-cad7-11e3-9281-0800271c8db1 has 15% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using smallestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 2faaa800-cad7-11e3-9281-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 2faaa800-cad7-11e3-9281-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 2faaa800-cad7-11e3-9281-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 288fe09e-cad7-11e3-9490-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 288fe09e-cad7-11e3-9490-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 288fe09e-cad7-11e3-9490-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 288fe09e-cad7-11e3-9490-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 2369b540-cad7-11e3-8fae-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 2369b540-cad7-11e3-8fae-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 2369b540-cad7-11e3-8fae-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 2369b540-cad7-11e3-8fae-0800271c8db1 and on Dock: 11
```

Figure 36: Screenshot of test 3c of experiment 2

The screenshot in Figure 36 shows that there are three resources. The smallest free resource first scheduler schedules the tasks *IntY1*, *IntY2*, *Demux*, and *Avg* on the first resource. The tasks *GenX2*, *SorU*, *SorX1*, and *SorX2* are scheduled on the second resource and tasks *UsIn*, *GenU*, and *GenX1* are scheduled on the third resource.

```
Resource with id: 6a57b8a2-cad8-11e3-8727-0800271c8db1 has 30% free CPU capacity
Resource with id: 7025c3c8-cad8-11e3-91cf-0800271c8db1 has 10% free CPU capacity
Resource with id: 79e4cb98-cad8-11e3-86f5-0800271c8db1 has 15% free CPU capacity
Resource with id: 830fc416-cad8-11e3-8859-0800271c8db1 has 30% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using smallestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 7025c3c8-cad8-11e3-91cf-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 7025c3c8-cad8-11e3-91cf-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 79e4cb98-cad8-11e3-86f5-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX2 scheduled on node: 79e4cb98-cad8-11e3-86f5-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 79e4cb98-cad8-11e3-86f5-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 6a57b8a2-cad8-11e3-8727-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX2 scheduled on node: 6a57b8a2-cad8-11e3-8727-0800271c8db1 and on Dock: 7
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 6a57b8a2-cad8-11e3-8727-0800271c8db1 and on Dock: 8
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY2 scheduled on node: 6a57b8a2-cad8-11e3-8727-0800271c8db1 and on Dock: 9
FAO: Sending a resource reservation request to the Resource Manager!
Component: Demux scheduled on node: 6a57b8a2-cad8-11e3-8727-0800271c8db1 and on Dock: 10
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 6a57b8a2-cad8-11e3-8727-0800271c8db1 and on Dock: 11
```

Figure 37: Screenshot of test 3d of experiment 2

The screenshot in Figure 37 shows four resources that are scheduled using the smallest free resource first scheduler. The workflow tasks *SorX1*, *SorX2*, *IntY1*, *IntY2*, *Demux*, and *Avg* are scheduled on the first resource. The tasks *UsIn* and *GenU* are scheduled on the second resource, and *GenX1*, *GenX2*, and *SorU* are scheduled on the third node. The fourth resource is not used for the scheduling.

```
Resource with id: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 has 25% free CPU capacity
Resource with id: 60582484-ceb8-11e3-98d4-0800271c8db1 has 10% free CPU capacity
Resource with id: 66f8a35e-ceb8-11e3-b119-0800271c8db1 has 15% free CPU capacity
Resource with id: 6f016612-ceb8-11e3-ba22-0800271c8db1 has 20% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using largestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 6f016612-ceb8-11e3-ba22-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 66f8a35e-ceb8-11e3-b119-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 6f016612-ceb8-11e3-ba22-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 and on Dock: 7
```

Figure 38: Screenshot of test using two applications of first FAO

The screenshot in Figure 38 shows the scheduling output of the first FAO of the test using two applications in the framework. The scheduler schedules the tasks *UsIn*, *GenU*, *SorU*, and *Avg* on the first resource, the task *SorX1* is scheduled on the third resource, and the tasks *GenX1* and *IntY1* are scheduled on the fourth resource.

```
Resource with id: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 has 5% free CPU capacity
Resource with id: 60582484-ceb8-11e3-98d4-0800271c8db1 has 10% free CPU capacity
Resource with id: 66f8a35e-ceb8-11e3-b119-0800271c8db1 has 10% free CPU capacity
Resource with id: 6f016612-ceb8-11e3-ba22-0800271c8db1 has 10% free CPU capacity
Submit workflow SimpleWorkflow
FAO: Start scheduling the workflow using largestFreeCapacityFirst scheduling

FAO: Sending a resource reservation request to the Resource Manager!
Component: UserInput scheduled on node: 60582484-ceb8-11e3-98d4-0800271c8db1 and on Dock: 1
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorU scheduled on node: 66f8a35e-ceb8-11e3-b119-0800271c8db1 and on Dock: 2
FAO: Sending a resource reservation request to the Resource Manager!
Component: GeneratorX1 scheduled on node: 6f016612-ceb8-11e3-ba22-0800271c8db1 and on Dock: 3
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterU scheduled on node: 5b53a2ec-ceb8-11e3-b1da-0800271c8db1 and on Dock: 4
FAO: Sending a resource reservation request to the Resource Manager!
Component: SorterX1 scheduled on node: 60582484-ceb8-11e3-98d4-0800271c8db1 and on Dock: 5
FAO: Sending a resource reservation request to the Resource Manager!
Component: IntersecterY1 scheduled on node: 66f8a35e-ceb8-11e3-b119-0800271c8db1 and on Dock: 6
FAO: Sending a resource reservation request to the Resource Manager!
Component: Averager scheduled on node: 6f016612-ceb8-11e3-ba22-0800271c8db1 and on Dock: 7
```

Figure 39: Screenshot of test using two applications of second FAO

The screenshot in Figure 39 shows the scheduling output of the second FAO of the test using two applications in the framework. There are four resources, with capacities equal to the starting capacities minus the capacities that are reserved for the tasks of the workflow of the first FAO. The scheduler schedules the task *SorU* on the first resource, tasks *UsIn* and *SorX1* on the second

resource, tasks *GenU* and *IntY1* on the third resource, and tasks *GenX1* and *Avg* on the fourth resource.