Final Report PG 428: Development of a Data- and Network Scheduling for Grid Computing Including a Flexible Evaluation Mechanism

Kay S. Brennecke, Stefan Einbrodt, Jan Philip Eumann, Sebastian Freitag,
Joern Gerendt, Manuel Heß, Bouchta Lakhal, Stefan Pinschke,
Zouhair Sabry, Daniel Sander, Sebastian Schlitte,
Thomas Wojczechowski

March 2004

Contents

Preface

The software covered in this document was created as part of the student working group PG428[1] at the University of Dortmund. This student working group consisted of 12 students of computer science at the University of Dortmund and was organized by Ramin Yahyapour, Carsten Ernemann and Volker Hamscher of the Computer Engineering Institute (CEI) which is part of the department of electrical engineering. The task was the development of a Grid Scheduler accounting for data- and network resources and including a flexible evaluation mechanism. Therefore the software was named **GridSched**, an obvious combination of the words "Grid" and "Scheduler".

Purpose of this document

As the title suggests this document is the final report of the student working group PG428. It explains the concepts as well as the features of GridSched in detail, including possible enhancements. To be able to understand the basic ideas and concepts the reader is expected to have basic knowledge in the areas of computer science and Grid Computing. However for a comprehensive understanding of all aspects of GridSched knowledge in the area of the Java programming language is recommended.

Structure

First an introductory part explaining what Grid Computing is about and which solutions currently exist is given in chapter 1. chapter 2 presents the basic ideas and concepts used in GridSched while chapter 3 provides a more detailed description of the several parts GridSched consists of as well as the features of the software. Due to the complexity and importance of the simulation chapter 4 has been designated to focus on all aspects of the simulation in detail. Chapter 5 covers features that had to be cancelled as well as possible enhancements. Finally chapter 6 explains how to install and configure GridSched. For an even more detailed description of the software the Javadocs of GridSched are available at `http://www-ds.e-technik.uni-dortmund.de/~pg428/`.

[1]PG is short for the German word "Projektgruppe". PGs for students of computer science at the University of Dortmund are numbered consecutively.

Chapter 1

Introduction

It is assumed that most of the readers have a basic knowledge of Grid Scheduling and its related terms and technologies. For reasons of completeness this introductory chapter gives the reader a brief introduction.

1.1 Motivation

Since the development of the first processor the computing power of available processors has increased dramatically. Nevertheless the research in proteins with specific properties to fight cancer, predictions of climatic changes and many other problems in science and industry obviously demonstrate the huge demand for computing power. Parallel computing systems and supercomputers are today utilized for these tasks. Comparing the growing complexity of computing tasks with the evolution of computers regarding their performance it is obvious that the increasing demand of computing power has not been met by an adequate amount of available computing power until today. Generally it is assumed that this correlation will not change in the foreseeable future. On the other side much potential computing power is wasted because of idle running. This leads to the idea to use a computer system's idle time to increase the total amount of available computing power.

This means there are basically two undesirable scenarios:

- the owner of a computer system has more computing power available than the executed computing tasks need,
- the owner of a computing task to be executed needs more computing power than available on the momentarily used computer system.

Additionally it is often very difficult to predict how much computing power a computing task will need. In the first scenario the computing power during the processor's idle time is unused and wasted. This results in an increase of the ownership's cost because the costs of purchasing and maintaining are spread on less computing power usage.

In the latter scenario computing jobs have to be postponed or canceled, or other computer systems coping with the computing task have to be found.

1.2 Meta Computing and Grid Computing

To decrease the costs of computing a more efficient computing power utilization is needed. The idea of Meta Computing respectively Grid Computing is to make computing power and resources available to others in a network environment. To the user it is presented as a virtual pool of computing resources providing computing power. Instead of looking at many computer systems manually for the needed amount of computing power and adequate kinds of resources a single system is used to execute the computing task.

The probably best example illustrating this idea is the Power Grid with its ease of use and its permanent availability. While the term Meta Computing refers only to computing power the term Grid Computing expands the term resource to all other possible resources. These may be resources like data or network connections. But one can also think of specialized resources like visualization caves, abstract resources like lecture halls or totally ordinary resources like coffee machines.
An example of a typical Grid Computing job may be a computation of data, the transfer of the results to a visualization cave, the modification of parameters, another computation on these data and a visualization of the final results on the same visualization cave. Due to latency problems and an exponentially high number of possibilities to combine different resources, job scheduling in a Grid is a complex task.

1.3 Existing Concepts

To accomplish Grid Computing industry and research institutions have worked on solutions. This chapter is intended to be a brief overview of the most important of them.

1.3.1 Globus Toolkit

The Globus Toolkit [19] is maintained by the Globus Alliance[8] which was founded in 1995. It includes code libraries and protocols which enable the management of distributed resources like memory and bandwidth. It's aim is to create a universal suite of protocols which are needed to build computational Grids just as the TCP/IP suite of protocols which enable communication between computers. The Globus Toolkit is based on web services which comply to the Open Grid Services Infrastructure (OGSI) [17]. The open architecture, the portability and the fact that decisions about standards are made in a community based forum, called the Global Grid Forum[7], are the main reasons why the Globus Toolkit is widely regarded as the project with the best chances to become a global and universal standard. The Global Grid Forum was founded in June 1999. It's main task is to support and to promote the development and establishment of Grid Computing technologies. This is accomplished by the documentation of 'best practices', technical specifications, experiments and guidelines. Global Grid Forum is the place where decisions about standards are made. Actually the prefix 'global' resembles reality regarding the fact that Global Grid Forum is joined by more than 5000 scientists, developers and users of more than 400 organizations from more than 50 countries.

1.3.2 Condor and Condor-G

Condor [11] has been developed for about 15 years by the Condor Research Project [4] of the University of Wisconsin-Madison. The goal is to "develop, implement, deploy, and evaluate mechanisms and policies that support High Throughput Computing on large collections of distributively owned computing resources". High Throughput Computing (HTC) environments are able to provide high amounts of computing power over long periods of time. Condor is a platform for users who want to provide unused computing power or computing power of dedicated servers. It enables the administrators of participating systems to decide under which conditions computing power is provided. In general Condor is used inside a single administrative domain. Condor-G [11] has been created to add the features of the Globus Toolkit to the Condor system. This means Condor-G is able to take advantage of security and resource access outside a single administrative domain. This is achieved by using the Globus Toolkit for inter-domain resource management. Simultaneously Condor is used for resource and job management within a single domain.

1.3.3 UNICORE and UNICORE Plus

UNICORE[5] is an abbreviation for "Uniform Interface to Computing Resources". It is a project which existed from 1997 to 1999. UNICORE and its successor UNICORE Plus (2000 to 2002) were funded by the German Federal Ministry of Education and Research. Several scientific institutions and international companies took part in this project.

UNICORE provides a software solution allowing users to submit jobs to remote systems without having to deal with details of the target operating system and other system's properties. This is achieved by using web based technology wherever it is possible. The user is able to edit jobs on her local computer before it is submitted to the UNICORE Grid. All these jobs may be monitored via the user's system.

1.3.4 EU DataGrid

The DataGrid[6] project was founded and is funded by the European Union. The goal is to create an infrastructure for the analysis and review of computer generated data for scientific experiments. The focus is on development and evaluation of infrastructures which support scientists working on their research, being independent on their location as long as they have a computer connected to the Internet at their hands.
The EU DataGrid project is lead by CERN but other international organizations are involved and take part in the development. More than 200 scientists participate in the DataGrid. This may ensure to bring forward the development of Grid technologies especially in aspects of data management.

1.3.5 Legion

Legion[1][14] is a project of the department of computer science of the University of Virginia and several partners. At this time it is the most advanced Meta Computing system and offers several testbeds.
It offers transparent access to distributed resources on different platforms appearing as one single high performance computer from the view of its users. For these distributed resources like computers, libraries, simulations and cameras it offers services as distribution strategy, data management, error tolerance and security features.

1.4 Summary

As the mentioned systems were created for different purposes they provide different functionalities. From the perspective of the user a system may provide a resource management which is not only limited to computing, network and data resources but includes all kinds of peripheral resources as well. The system may offer the ability to submit, monitor and control her computing jobs.
From the administrators point of view the system may provide the possibility to choose how to offer its resources. Furthermore security and billing issues have to be regarded. At this time no solution exists to cover all aspects of Grid Computing.
Comparing the mentioned systems shows some of them have an experimental character focused on specific research issues like the EU DataGrid which is focused on data management. Other systems try to fulfill the needs of a specific user group. For example Condor is designed to fit to the special needs of High Throughput Computing.
The UNICORE project as well as the Legion project helped in the Grid Computing research by finding solutions to a number of problems. But they are not designed to create a global standard.
The Globus Toolkit seems to be the project being suited to create a global standard for Grid Computing. It is based on global standards and provides the low level framework which is necessary to ensure interoperability between computing systems taking part in a Grid Environment.

Chapter 2

Core Concepts of GridSched

This chapter gives an idea of the essential concepts of the GridSched system. It describes the fundamental functional demands which are realized in the GridSched project. Basically these demands are fulfilled by the GridSched's scheduling process and its management of resources and reservations. The objective of this chapter is to summarize the essential ideas of the GridSched system developed during the analysis, design and prototyping phase of the GridSched project.

When GridSched is running in a bounded Grid Computing Environment it is called a **managed Grid Environment** or **managed Grid**. This means that the GridSched system takes control of only some or all available Grid Resources in this bounded address space which constitute a network domain as well. In a managed Grid only some or all GridSched services and components may be installed. Therefore the range of GridSched's control is fully decided by its carrier.

In GridSched there are two different groups of users involved: the resource providers and the users

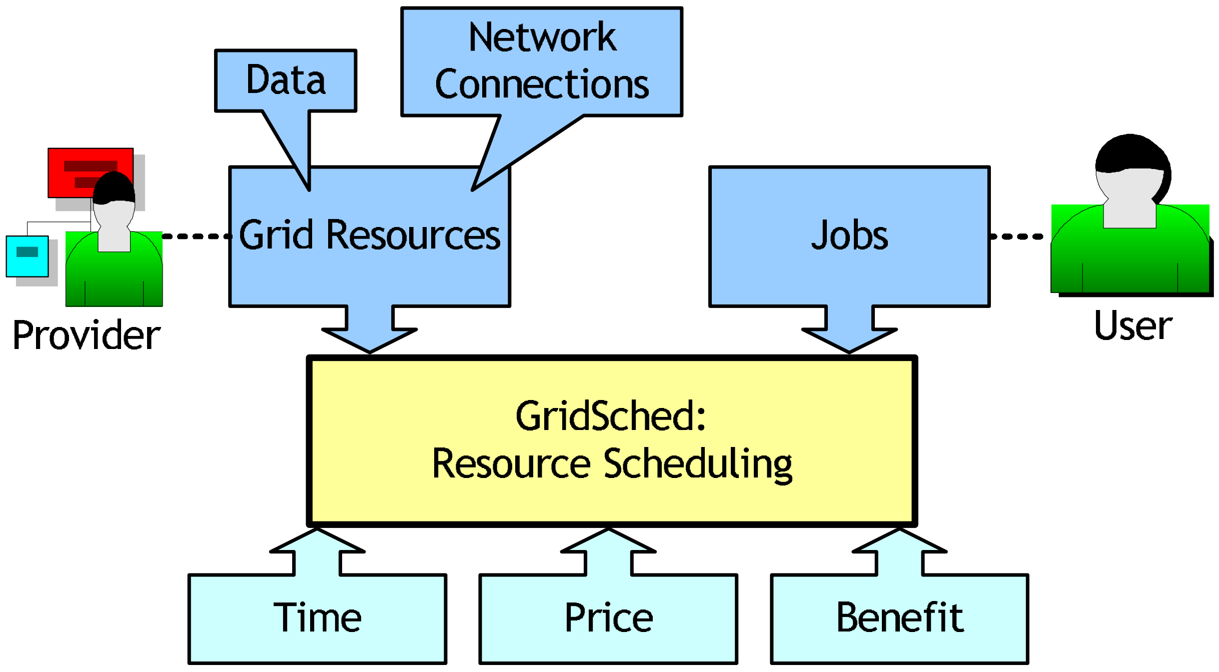


Figure 2.1: Main terms in GridSched's concept.

who want to execute jobs. The resource provider installs and starts the appropriate GridSched services to make her resources available in the managed Grid. Moreover she defines prices and policies to control the user's access to the resources. A GridSched user has to identify herself. Afterwards she is able to define jobs which may be executed if the appropriate resources are found. These correlations are shown in Figure 2.1.

2.1 Distributed Grid Scheduling

The GridSched developer's idea of Grid Scheduling is to combine Grid Resources and computing tasks regarding the aspects of time and benefit. The resulting approach is a middleware platform to unite the providers and users of distributed resources in a Grid Computing Environment.

GridSched's task is to execute computing jobs in a given period of time using specified Grid Resources which are under control of GridSched. In most of the existing Grid Computing Systems the idea of a Grid Resource is limited to common hardware items like CPU's and RAM. In GridSched the term of Grid Resource is extended by data and network connections. Actually GridSched's concept of a resource is not restricted to physically existing resources. In GridSched any act of providing a service is seen as a resource.

Large amounts of distributed resource in a Grid Computing Environment require a considerable effort

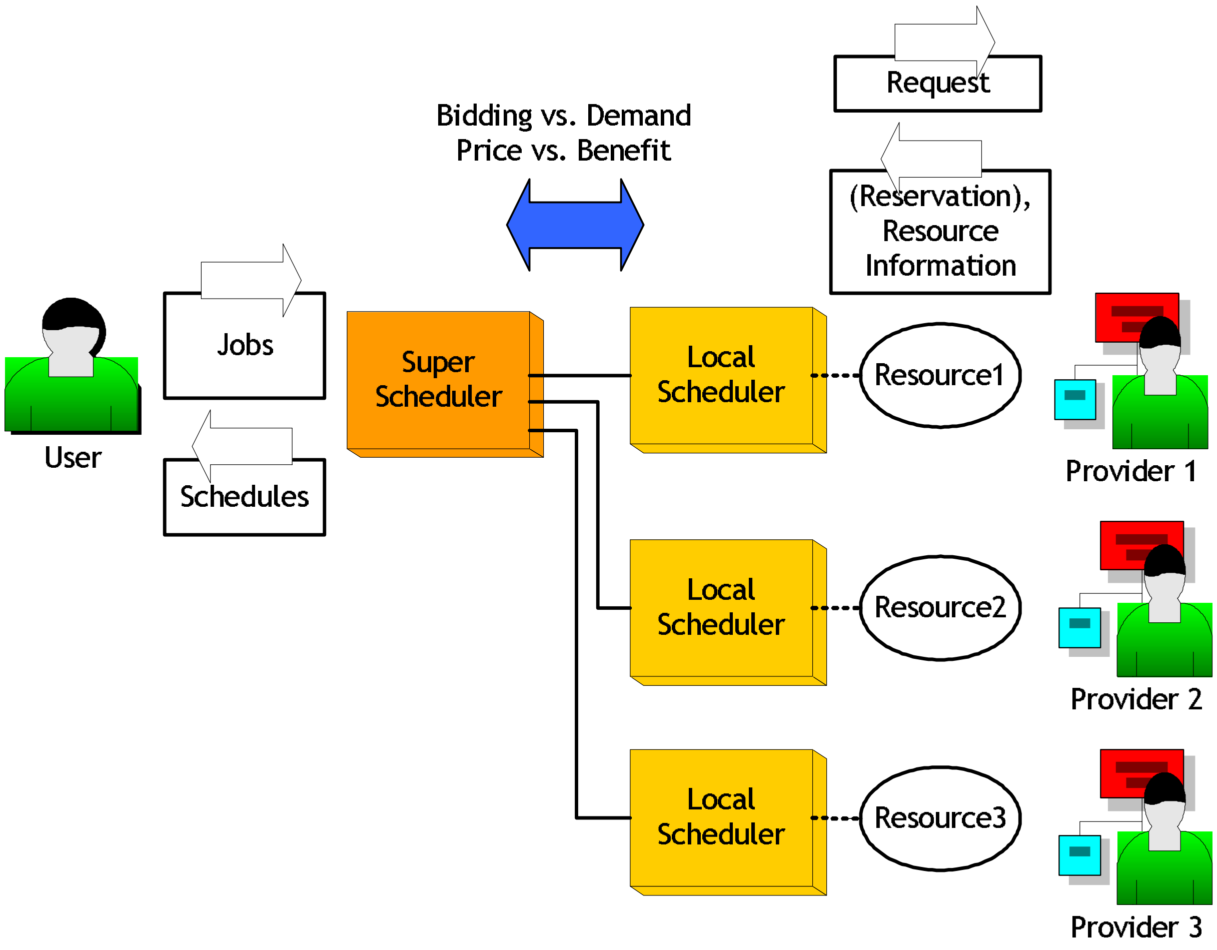


Figure 2.2: GridSched's concept of distributed scheduling.

concerning the management of resource information and reservation requests. A centralized management service is unsuitable to fulfill this requirement because of the enormous efforts to realize scalability.

Accordingly the concept of distributed scheduling provides a better solution. GridSched's essential parts of the distributed scheduling system are the GridSched's services **Local Scheduler**, **Super Scheduler**, **DataManager** and **NetworkManager**. All these executable components are accompanied by other executable and non-executable GridSched components which support the scheduling and reservation process indirectly.

The sum of these fundamental ideas makes up GridSched's approach. It is completely different to any other system existing at the time the project started.

2.1.1 Super Scheduler

In the context of Distributed Scheduling the Super Scheduler is to be seen as a coordinating function unit and as a counterpart to a cluster of Local Schedulers which are assigned to distributed resources. One of the two main tasks of GridSched's Super Scheduler is the processing of incoming jobs which have to be executed in a limited period and which are needing specified Grid Resources.

The other task is to generate reservation requests regarding the time and resource aspects of the job. Basically it is distinguished between binding and unbinding reservations. Unbinding reservations are applied during the process of determining a variety of possible resource schedules. After selecting a single schedule the binding reservations are used to take control of the specified resources.

In a Grid Computing Environment managed by GridSched it is possible to set up one or many instances of the Super Scheduler. In the GridSched project the Super Scheduler is called **Titan**.

2.1.2 Local Scheduler

GridSched's Local Scheduler is allocated to a specific computing resource offered in the managed Grid Computing Environment. It is needed to set up one instance of the GridSched's Local Scheduler on each hardware resource which has to be controlled by GridSched.

Basically it works on top of local hardware schedulers. GridSched's components are not able to perform hardware reservations or any other hardware handling. It is assumed that this functionality is provided by the hardware layer implementation of the Grid Computing Environment which is managed by GridSched.

One of the main tasks of GridSched's Local Scheduler is to manage and process incoming reservation requests received from Titan, GridSched's Super Scheduler. Besides GridSched's Local Scheduler stores qualitative and quantitative information on the managed resource in the directory service which is described in Section 2.1.3.

In further context GridSched's Local Scheduler is called **Bob**.

2.1.3 Directory Service

To find appropriate resources the directory service is needed. It stores all properties of the resources which exist in the managed Grid Computing Environment. A set of parameters describes the attributes of a resource.

The directory service is realized by using a central or distributed database system adapted to the demands of network management tasks. Basically every GridSched service involved in the scheduling process is able to access the directory service via **Lightweight Directory Access Protocol (LDAP)**.[1]

There is only one instance of the directory service needed in a managed Grid Domain.

[1]LDAP is a standardized protocol for using directory services. It is independent from manufacturers and is available for a big variety of platforms. It provides the access to a directory and defines its hierarchical data structure and a name space. For more information on using LDAP in context of Grid Computing see [16].

2.1.4 Schedules

The result of the scheduling process is represented by one or many resource schedules. Resource schedules are generated by Titan processing the given job and the answers to reservation requests sent by GridSched's Local Schedulers. A schedule represents a combination of resources to perform a given job regarding the resource's time of availability.

If a valid schedule is retrieved Titan tries to reserve the scheduled resource. Therefore it uses the same communication path used for the reservation requests. Afterwards the regarding job is executed.

In GridSched the resource schedule is supplemented with an entity representing costs. It is assumed that the usage of resources will cause costs for the providers. Moreover the users of resources have to pay a price when using the resources.

To limit the complexity of the GridSched project a billing system is not implemented. All costs and prices are meant to be symbolic.

2.2 Evaluation of Schedules

To take account of all different aspects of Grid Resource usage the cost respectively the degree of productivity of a resource is combined with its time of availability and its quantitative parameters. The degree of productivity is represented by the so-called **ObjectiveFunction**. It is given for a specific combination of resources and is determined by a mathematical function which is an algorithmic description of the user's preferences regarding qualitative and quantitative resource parameters. An **ObjectiveValue** is a specific value of the ObjectiveFunction. It is calculated by Titan when determining a valid schedule for a given job. The size of the ObjectiveValue is proportional to the size of productivity of a resource combination respectively to the price of the resource's usage.

GridSched's user can define a limit for a price. This is the criteria to choose one of many schedules determined by Titan. Moreover it enables GridSched's users to put a focus on resources used for their computing jobs. Generally this means a user can have different point of views when evaluating a schedule:

- job execution on specialized Grid Resources to maximize productivity regardless of the costs,
- fast job execution by using powerful but expensive Grid Resources,
- inexpensive job execution by using less powerful but inexpensive Grid Resources,
- inexpensive job execution by accepting long durations because of using idle resources in several distributed intervals.

On the other side a resource provider may have totally different aspects in mind regarding the evaluation of a schedule:

- profit maximization by offering resources at a high price,
- load maximization by offering resources at a low price during periods when they are used at a low rate,
- satisfaction of the resource user regardless of cost consumption or prices which actually may be achieved.

So the fundamental part of GridSched's distributed scheduling concept may be understood as a market-based negotiation process between resource providers and resource users. As mentioned before GridSched is able to determine prices and costs but it does not provide any billing functionality which may be an item of future enhancements.

2.3 Scheduling Strategies

All scheduling strategies are represented by algorithms which are implemented in scheduling modules. Two strategies to find a valid schedule are presented in detail in Chapter 3. A scheduling module may be replaced by another scheduling module. Thus GridSched's Super Scheduler may be adapted to the specific needs of GridSched's carrier[2].

2.4 Conceptual Aspects

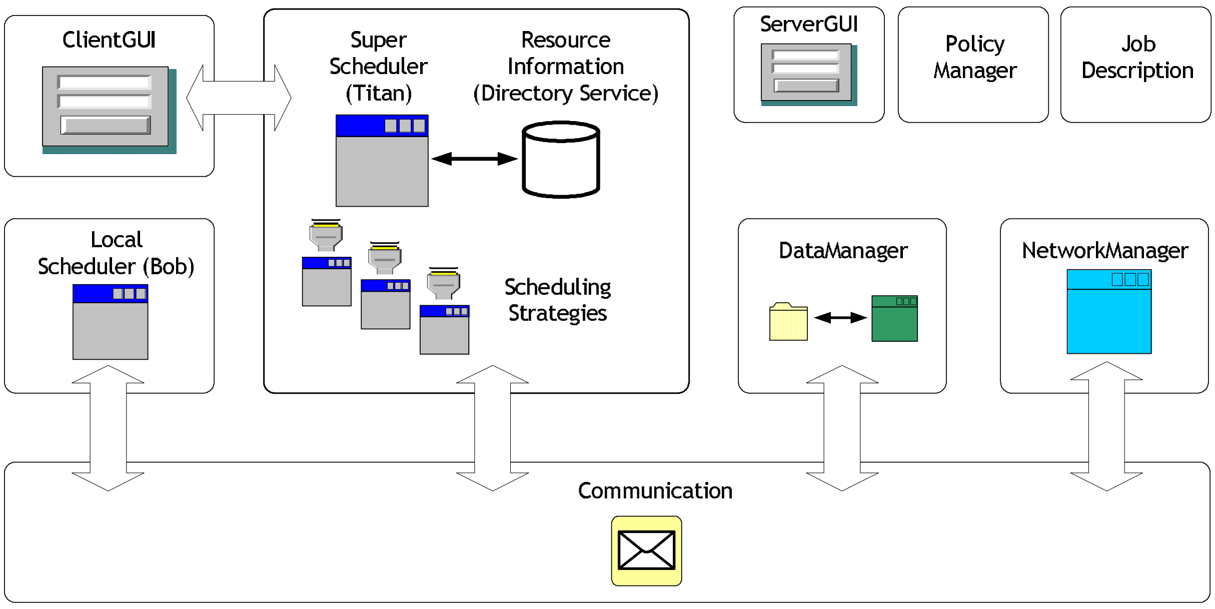


Figure 2.3: A view on GridSched's components.

Besides all aspects regarding resource scheduling, there are some further important problems which need GridSched's solution.

2.4.1 Hardware Management

As mentioned before the hardware resources have to be managed and reservations must be handled. This is part of GridSched's Local Scheduler called **Bob** which works on top of existing hardware resource schedulers. This special type of scheduler is a part of the Grid Computing Environment managed by GridSched. Most of the management tasks of Bob have been explained in Section 2.1.2.

2.4.2 Network Management

Network connections are treated like a special type of resource. Therefore a distinctive GridSched component is needed for management tasks. In GridSched it is called **NetworkManager**.

Basically this component determines what network connection routes are generally available in a managed network domain. When receiving a reservation request it is examined if a connection between the given start and end point is available.

[2] A carrier is a person or an organisation which uses GridSched to provide job execution on a Grid Computing Environment to GridSched's users. A carrier may be a resource provider as well.

Like GridSched's Local Scheduler the reservation is done by the hardware layer implementation of the Grid Computing Environment managed by GridSched. Analogous to GridSched's Local Scheduler the NetworkManager of a network domain handles all incoming network reservations but it does not execute them.

Besides the period of availability other important properties of a Net Resource are handled by Titan and NetworkManager: latency, bandwidth and **Quality of Service (QoS)**.

QoS may be described as a size to define the assurance of the mentioned network connection properties like period of availability, latency and bandwidth. Special network services like **DiffServ** or **MPLS** provide QoS[3]. They are embedded into network's hardware components like switches and routers to implement network reservations.

Another important outlook is the processing of network routes which cross the boundary of a network domain managed by a single NetworkManager. As mentioned before one instance of the NetworkManager controls a single network domain. Thus all instances of the NetworkManager must communicate and coordinate with each other to handle multi-domain network connections.

2.4.3 Data Management

Data is seen as a special type of resource and it appears in the form of files in an operating system's file system. These files are objects processed by GridSched's DataManager. All files used in a managed Grid Environment are registered in the directory service by the DataManager. As they are registered they may be queried by any GridSched's scheduler.

GridSched's DataManager is able to transfer a data file from its source location to a distant location in the managed Grid Environment. The destination is a computer needing the particular data file to execute a computing job in a given period. To execute a copy job GridSched's DataManager uses GridSched's NetworkManager.

For scheduling purposes it is necessary to consider the following situations regarding copy jobs:

- the file to be processed is available at a specific location in the managed Grid Environment,
- the file to be processed is available at several different locations in the managed Grid Environment,
- The source and destination locations are not connected by a network route,
- The source and destination locations are connected by one or many network routes. In the case that more than one route is available these routes differ in their characteristic properties like duration of availability, bandwidth and QoS.

To schedule the execution of copy jobs it is important to consider the amount of memory space available at the destination location.

2.4.4 User Access Management

Not only for Data Resources it is important to control user's access to resources managed by GridSched. To access resources which are managed by GridSched the authorization of GridSched users is needed. This functionality is provided by a GridSched component called **PolicyManager**.

With PolicyManager it is possible to define rules which describe the permitted or forbidden access to specific resources for users or user groups.

[3]For more information on DiffServ, MPLS and other QoS concepts in context of Grid Computing see [3]

2.4.5 Job Description

The definition of a job contains the description and enumeration of the needed resources. GridSched's job description may be seen as a computing language algorithmically interpreted by the system. This is done by declaring a period of time, the amounts of needed resources and all parameters describing their qualitative characteristics like processor performance, processor type or existence of a co-processor. Moreover the chronological and spatial dependencies between single Grid Resources may be expressed.

To realize the mentioned market-based negotiation process the number representing costs respectively the degree of productivity is integrated into a job description. A maximum price given by a GridSched user is also deposited in the individual instance of a job description.

A job description is used globally in GridSched. It represents a data structure to process jobs and resulting reservation requests universally in GridSched. Its hierarchical characteristics allows to compose a job by defining separated sub-jobs.

2.4.6 Job Definition

According to the syntax of GridSched's job description language a job is defined by using a Graphical User Interface (GUI). In GridSched it is called **ClientGUI**. The name of the GUI points out the role of its users in GridSched.

All items are accessed by editable text fields and drop-down boxes. This GUI also gives information on all user's defined jobs and its state. GridSched's users have to authorize themselves to access the system. A user is only allowed to see jobs she defined by herself.

2.4.7 System Control and Configuration

Most of the GridSched's components are executable services which are controlled via a unique GUI named **ServerGUI**. Its functionality includes starting, stopping and configuring of GridSched's services.

If the GUI and the services reside on the same machine the service control may be done locally. Alternatively the service control may be executed remotely if the GUI and the services reside on different machines.

Chapter 3

Design and Implementation of GridSched

3.1 Introduction

This chapter focuses on the design and the technical realization of GridSched. At first an explanation of what is understood to be the different roles of participants in Grid Scheduling is given. Then it will elaborated on which components were designed to support those participants in acting the roles identified and how these components interact in order to form a complete Grid Scheduling solution.

3.1.1 Roles in Grid Scheduling

The Grid is a collection of sites which are connected by a common, heterogeneous network. Two roles that each site may act as were identified. They are defined as follows: (Also see Figure: 3.1.1)

- **Provider** The provider offers the service of accessing some of its resources to the Grid.
- **User** The user accepts the services of the providers to perform operations which she couldn't perform with her resources alone.

At this point it is necessary to define what a resource is.

3.1.1.1 Resources

A resource might be just about anything that meets the following criteria.

- It is connected to some kind of service which can be offered to the Grid.
- There are attributes to distinguish the properties of this resource from others.
- The acceptance of that service is limited to a specific span of time, a **time slot**. A time slot begins at a certain time, goes on uninterruptedly for a specific duration and ends after that duration is over.

Many kinds of services fulfill these criteria. Consider a room-rental-service where the usage of a resource (the room) which is specified by its location (attributes) can be booked for some span of time (time slot). A service would usually not be exclusively offered to the Grid but may be accepted by other parties as well. In the case of a room-rental-service a room may possibly belong to a hotel where it can also be booked by phone or otherwise. So there are usually multiple systems that organize access to one resource.

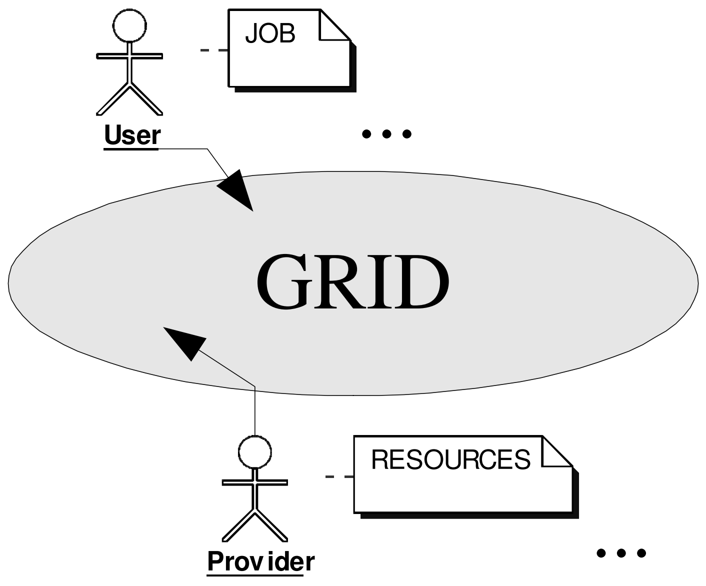


Figure 3.1: Roles in Grid Scheduling.

A scheduling system for the Grid must take into account that it will most likely not have exclusive control over a resource, but must share it with other scheduling systems.

Of course each site may act more than one role. One site may provide access to some of its resources while it needs to access other resources that it doesn't possess itself. In that case, the site acts as a provider as well as a user.

3.1.2 Strategy and Components of GridSched

In the case of GridSched it is differentiated between two kinds of resources which might best be named '**physical**' and '**virtual**' resources.

- A physical resource exists regardless of it being used or not. This applies for resources like computer hardware or for rooms as described in the example above. Reserving such a resource is achieved by marking the resource as reserved for the user, no further actions have to be taken.
- A virtual resource does not exist as long as it is not used. Reserving such a resource usually involves reserving other resources which may themselves be either virtual or physical. Consider the following example.
  A user wants to process a digital file on a computer-system. To achieve this the following services are needed.
  - **The computer** Of course access to the computer is needed. It is a physical resources since the computer exists even if it is not accessed.
  - **The File** The file itself exists even if it is not used, but it does not necessarily exist on the desired machine. So the service of accessing a file is a virtual resource which may consist of multiple other resources:
    * **The computer** Again, a computer, or more precisely its mass storage devices are needed to store the file for the time it is meant to be accessed.
    * **The file-transfer** If the file does not yet exist on the target computer, it must be transferred. This is a virtual resource because the file is only transferred when needed. Again transfer of a file needs other resources to be successful:

∗ **The network** While a network itself is a physical resource, the service of routing packets over that network is not. In case of a true reservable network connection, a dedicated route between the source and the target site must be established which only transports packets of the user who reserved that connection.

To reflect the roles and the kinds of resources identified the following structure was created for GridSched (See Figure: 3.1.2).

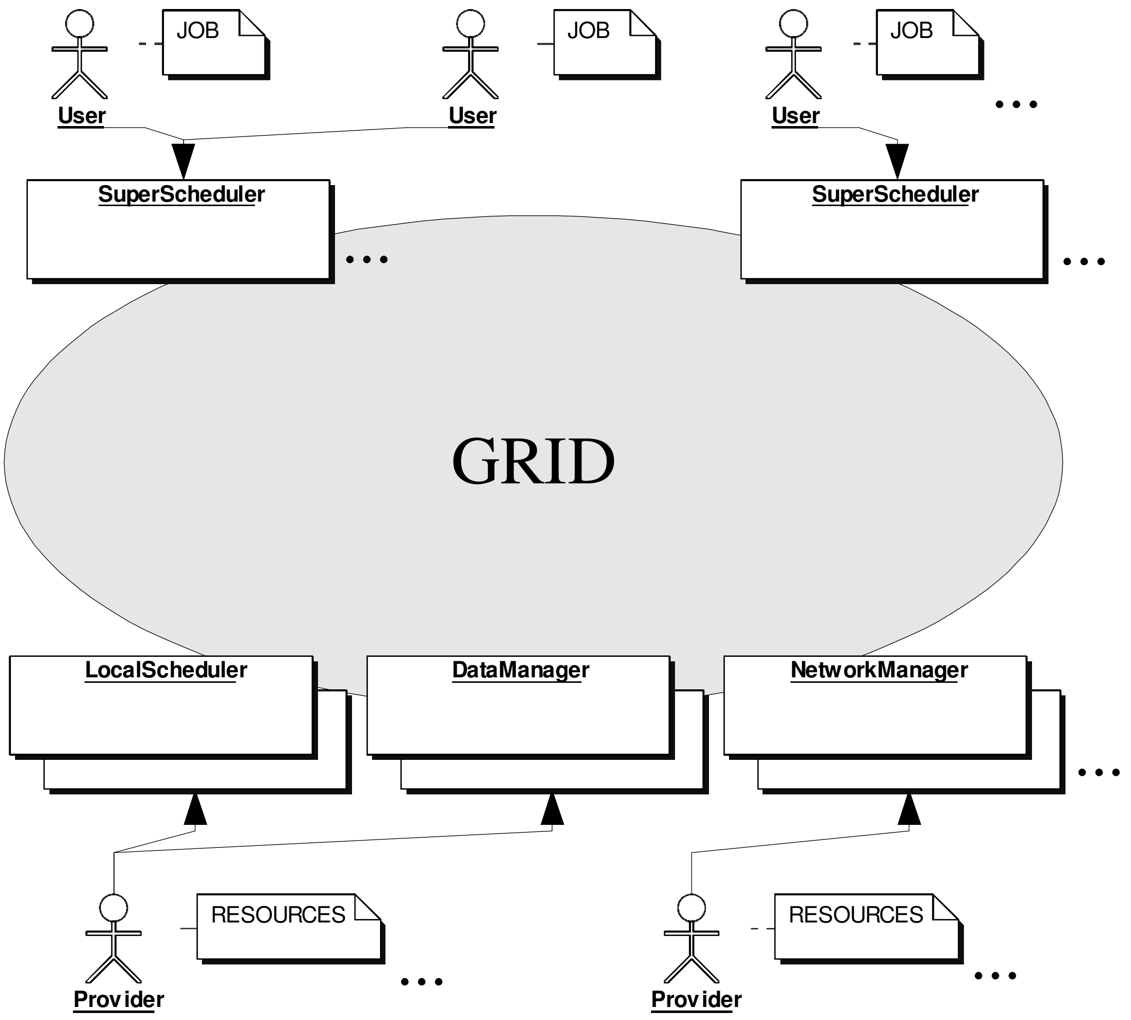


Figure 3.2: Architecture of GridSched.

- **The role of the provider** Providers should be rendered capable of offering the services of their resources to the Grid. This includes among other things: publishing the resource's existence, interaction with local scheduling systems and access control.
  For each kind of resource covered by GridSched an independent piece of software was created to provide these features. See Local Scheduler(Section: 3.6), DataManager(Section: 3.7) and NetworkManager(Section: 3.8) on details about these components.
- **The role of the user** Users should be enabled to find and assemble resources of the Grid in such a way that they can be employed in a job. To accomplish this a software is needed that can access the Grid to find resources that reflect the users requirements and to produce a combination of these resources which is capable of processing the job. This part of GridSched is called a **Super Scheduler** which is described in (Section: 3.4).

For being able to deal with any kind of resource in a flexible manner a common communication interface was created that Super Schedulers and Resources employ for interaction. Based on the definition of resources (Section: 3.1.1.1) the information that needs to be exchanged can be reduced to the following:

- **DISCOVERY** It must be possible to learn about the existence of a resource in order to use it. Over the course of the following sections, this kind of interaction may occur in the name of 'getting resources'.
- **AVAILABILITY** It is necessary to verify that a given resource is available to the user at a given time. Over the course of the following sections, this kind of interaction will be referred to as 'getting timeslots'.
- **OFFER** It must be possible to learn about the concrete circumstances under which the resource can be used. This includes agreeing on a precise start- and end-time of the service that is to be accepted. Over the course of the following sections, this kind of interaction may be referred to as 'getting offers'.
- **NOT_BINDING_RESERVATION** A reservation of a resource is understood to be the act of agreeing to an offer previously issued by the provider of a resource. By reserving a resource it is ensured that for the given span of time, the associated service is available to the user. There are two levels of reservation as part of a two phase reservation procedure: It must be ensured that all resources employed for a job can successfully be reserved or that none of those resources is reserved. Therefore it is demanded that each resource offers the possibility of performing a reservation which is valid for a limited span of time only. This span of time serves as a grace period within which all other resources needed can be reserved. Only if all resources can be successfully reserved in this way, the second phase of reservation can commence. Over the course of the following sections, this kind of interaction may occur as 'reserving gracefully'.
- **BINDING_RESERVATION** While a "not_binding_reservation" simply prevents that another party performs the same reservations while another reservation is in progress, a binding_reservation takes full effect and effectively guarantees that the condition reflected by the properties of the previously issued offer holds. In other words, after a binding_reservation the service as described by the offer is available to the reserving user, and all consequences of this reservation take effect.

  One reason why this is important is that the act of reserving a resource incurs demanding a fee from the reserving user for the acceptance of the associated service. A job however consists of many of such reservations. If only some of those reservations can be done successfully, the job still can not be carried out, but the reservation fees for the successful reservation would nevertheless take effect. Over the course of the following sections, this kind of interaction may be referred to as 'reserving bindingly'.
- **CANCEL_RESERVATION** The reservation of a resource may be cancelled. The conditions under which this is possible may vary according to the kind of resource and to the providers' policies. Cancelling a resource means that the associated service will no longer be available to the reserving user, but there is no guarantee that an associated reservation fee will be re-transferred to any extend.

This small set of access methods makes it possible to create a very abstract view of the Grid and to present any kind of resource in a unified way.

3.2 Graphical User Interfaces

GridSched's component called **GUI** is made up of three different graphical user interfaces. Each serves a specific purpose and user group.

The part of GridSched's GUI called **Client GUI** is used to define and manage jobs to be executed in the Grid Computing Environment managed by GridSched. Its name expresses the role of its users within the GridSched system.

Server GUI is the part of GridSched's user interface component which utilizes administrative features of GridSched to its carriers or administrators. This user interface provides control of all GridSched services running in the managed Grid Computing domain.

As shown in Chapter 5 GridSched is supplemented by a simulator component. A specific user interface is needed to handle this executable part of the GridSched project. It is called **Simulation GUI** and is assigned to GridSched's user interface component, but functionally it is part of GridSched's simulator. Thus it is described in Chapter 5.

3.2.1 ServerGUI

GridSched's ServerGUI is used to control and configure all GridSched services running in the managed Grid environment. The name of this GUI describes the part of the GridSched system which is accessed via this interface. It consists of two individual parts called **Service Control GUI** and **Service Configuration GUI**.

All individual parts of the ServerGUI are created dynamically at runtime. As mentioned before in a Grid environment managed by GridSched its types of service can exist as one or many instances. But it is also possible that one or many types of service does not exist. So in Service Control GUI only those control elements are provided that correspond to actually installed GridSched services.

For each installed service a separate Service Configuration GUI is provided to change all available configuration items. To the GUI's user it seems that each installed service has its own individual Service Configuration GUI. Actually there exists only one instance of the Service Configuration GUI which is built up individually each time the user wants to change a service's configuration. So all parts of GridSched's ServerGUI which are specific for the configurations of services are built up dynamically.

Service Control GUI

Requirements Starting and stopping GridSched's services and accessing their configuration parameters is the sum of functionality included in this part of GridSched's ServerGUI.3.3 This is provided for local GridSched services running together with the Server Control GUI on the same machine and for GridSched services running on different machines far off but within the managed Grid domain.

This means that local and remote GridSched services are controlled via a single user interface. For this reason the IP Address or the name of a remote machine is entered into a specific text field and all installed GridSched services on the selected machine are determined. This is done by checking the existence of the corresponding configuration file. So the Service Control GUI is built up dynamically for a specified GridSched server.

Architecture The architecture of the Service Control GUI consists of the **Swing** container class[1] and a control class named `ServiceControl`.3.4

The object derived from `ServiceControl` starts and stops GridSched services. It also calls the Server Configuration GUI using a service identifier as the parameter.

[1]Swing is part of the Sun Java language to provide platform-independent GUI programming. It defines a huge variety of GUI control elements. A Swing container class encapsulates Swing control elements like buttons or text fields to place them into a GUI frame. Many Swing classes are inherited from the **Abstract Windowing Toolkit (AWT)** classes. AWT is the predecessor of Swing. [12]

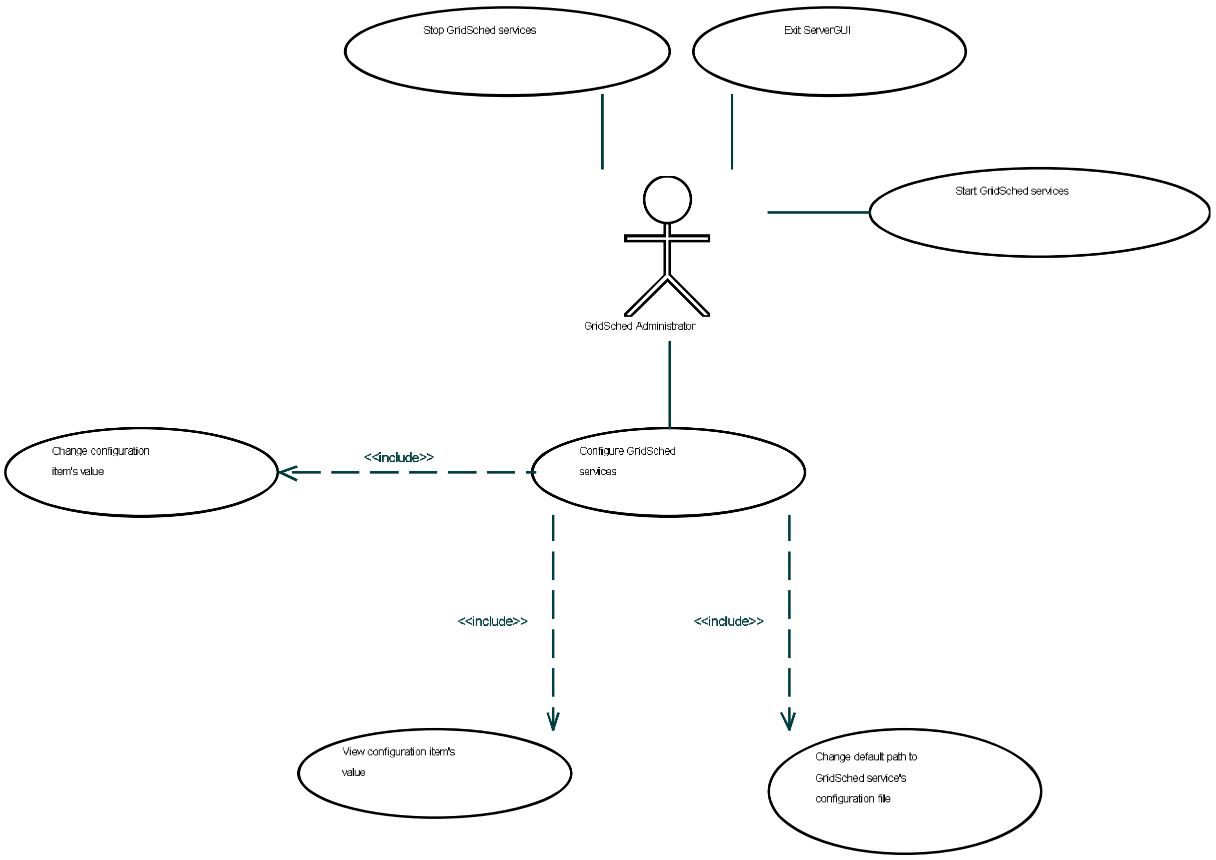

Stop GridSched services
Exit ServerGUI
Start GridSched services
GridSched Administrator
Change configuration item's value
<<include>>
Configure GridSched services
<<include>>
<<include>>
View configuration item's value
Change default path to GridSched service's configuration file

Figure 3.3: Use Cases of GridSched's ServerGUI.

The `ServiceControl`'s constructor determines all locally installed GridSched services by checking the existence of the corresponding configuration file. The services respectively the corresponding configuration files and the `ServiceControl` object have to reside on the same machine. So the `ServiceControl` object manages only GridSched services which are locally to it. The command to execute the mentioned control functions is received from an object derived from `ServerGUI`.

Coincidentally a `ServerGUI` object represents the event listener of all control elements contained in `ServerGui` such as the "Start", "Stop" and "Configure" button and the ComboBox to select a GridSched Server. When these control elements used by the GUI's user an event is fired. All events are processed by the event listener object using corresponding methods to start and stop services or to build up the Service Configuration GUI. Using **Remote Method Invocation (RMI)** it is easily possible to realize remote control on `ServiceControl` objects residing far off the machine on which the `ServerGUI` object is residing.

Entering a new IP address or a new name into the combo box causes an event which triggers a method to build up the contents of the `ServerGUI`'s container newly. Calling a unique method of the `ServiceControl` object the `ServerGUI` object gets the identifiers of all locally installed GridSched services.

Service Configuration GUI

Requirements This part of GridSched's ServerGUI makes it possible to edit single configuration items of a GridSched's service.3.3 It is called by the Service Control GUI.

Before starting the Service Configuration GUI the path of the corresponding configuration file may

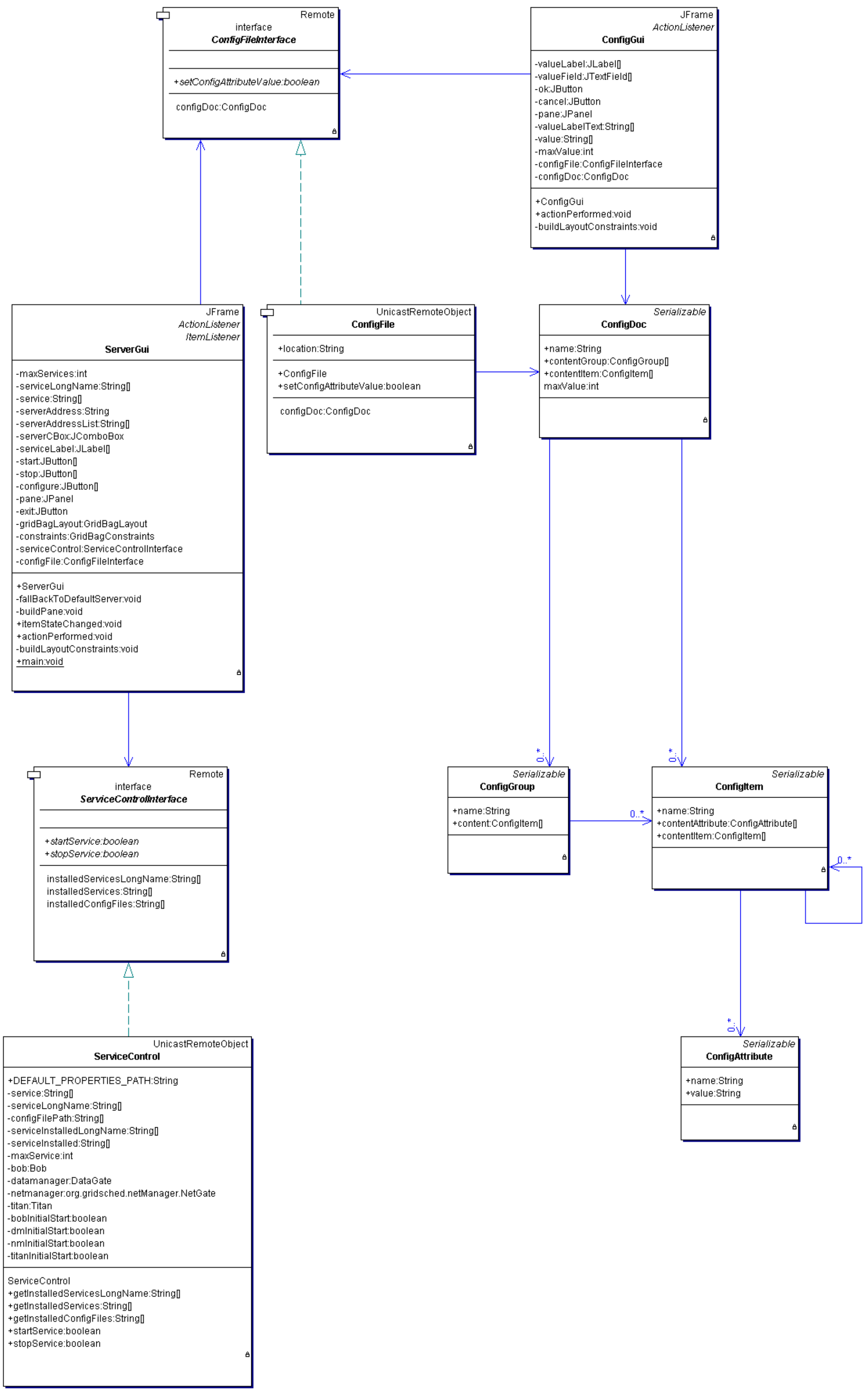


Figure 3.4: Classes of GridSched's ServerGUI.

be changed via a dialog. All available parameters and their values are read from a configuration file and presented in a dialog. This dialog is built up dynamically. So only one Swing container class is needed to create to edit all existing configuration files. This effects that the code is easy to maintain. Changes may be done very quickly for all kinds of configuration file representations.

All configuration parameters are ordered hierarchically in a XML document stored in a configuration file. Some of the parameters are grouped together in hierarchy items. The hierarchy is visualized in the Service Configuration GUI's tree view. All parameters and their values contained in a hierarchy item may be viewed and edited in text fields contained in a detail pane which is shown when selecting the corresponding hierarchy item.

Like the Service Control GUI an object derived from the Service Configuration GUI class represents an event listener as well. The OK button fires an event that causes the parameters to be written into the corresponding configuration file.

Architecture A configuration file stores configuration information as a XML document. This document respectively the contained configuration parameters are read and processed by an object derived from the `ConfigFile` class and from the objects representing GridSched's services. 3.4 The structure and the contents of the XML documents are static and given by the service's developers.

To process the XML documents inside the Service Configuration GUI it is transformed into a object-based hierarchical data structure. A single XML document is represented by a `ConfigDoc` object. This object can contain one or many `ConfigGroup` objects and `ConfigItem` objects. These objects represent containers for `ConfigItem` objects.

A `ConfigItem` object contains `ConfigAttribute` objects representing the configuration parameters. Like the parameters a `ConfigAttribute` has an identifier and a value. All other objects exist to order and group `ConfigAttribute` object. They represent the structure of an XML document. Like `ConfigAttribute` objects they have identifiers to store the XML element names but no values.

A `ConfigDoc` object is always assigned to a `ConfigFile` object. A `ConfigFile` object provides methods to read and write a configuration file. In this methods a `ConfigDoc` object is a parameter or the returned value. Similarly to the distribution of work of `ServerGUI` and `ServiceControl` objects `ConfigFile` objects have a controlling task and they are used by `ConfigGUI` objects. The whole process of reading and writing a configuration file is illustrated in the Sequence Diagram in 3.5

Implementation Details

The implementation of ServerGUI is using Java's Swing and AWT classes to implement both graphical user interfaces especially for the interface's containers, control elements and the event listeners.

To realize the remote access to distant objects Java's RMI is utilized.
`ServiceControl`'s constructor method creates a remote type of a `ConfigFile` object for each existing configuration file. Afterwards these remote objects are registered at a central RMI Registry. The `ConfigFile` objects and `ServiceControl` object must reside on the same machine. A `ServiceControl` object itself is instantiated and registered by the `ServerGUI`'s main method which is the main method of the executable ServerGUI component as well.

Stopping a GridSched service by using the Service Control GUI interrupts the execution of the service but it does not terminate the existence of its instance. Thus a new start of a service via the GUI after changing its configuration parameters does not restart the service with the changed configuration. This has to be done by killing the service's process via the operating system of the local host before pressing the GUI's "Start" button.

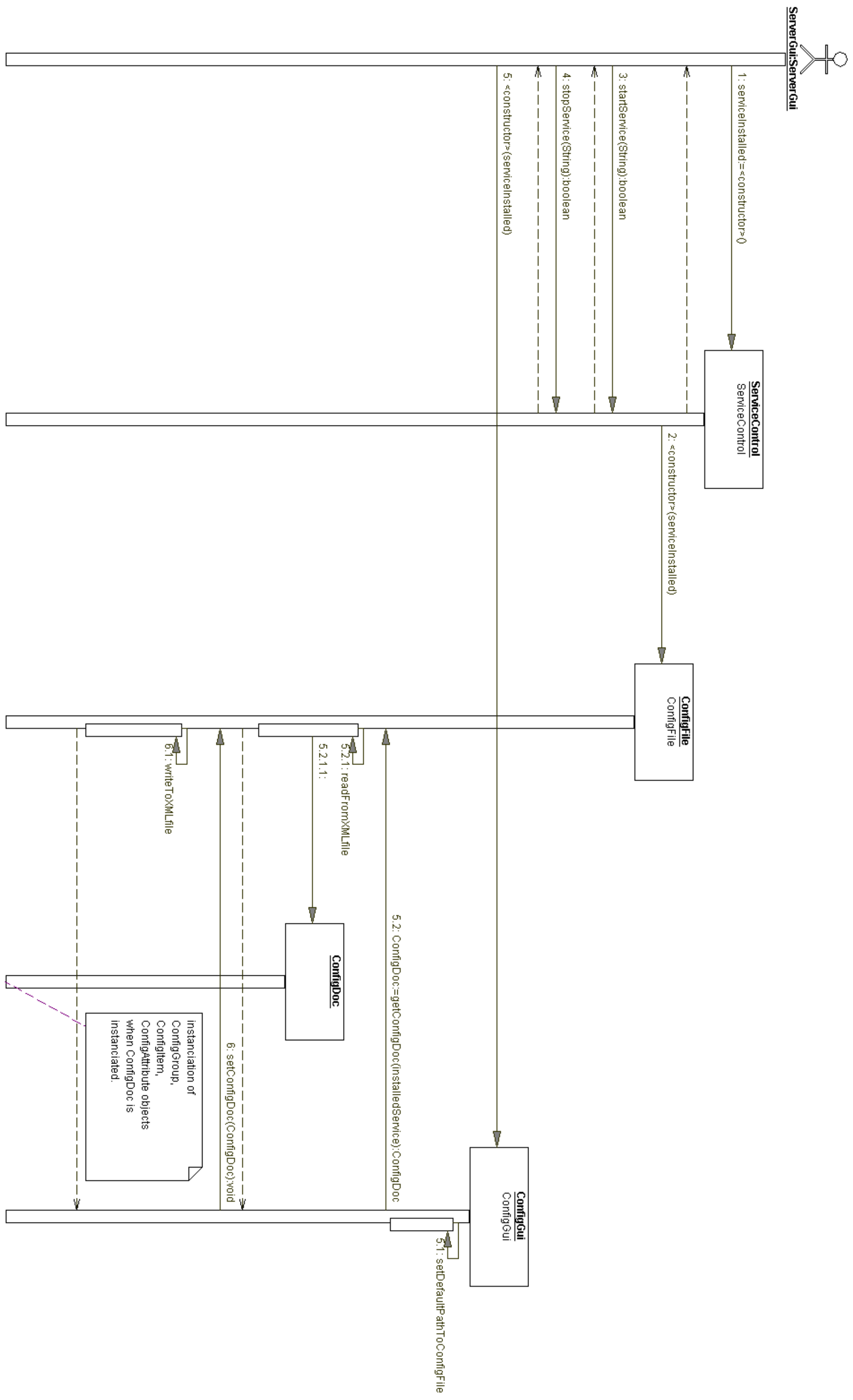

ServerGui:ServerGui
1: serviceInstalled:=<constructor>()
ServiceControl
ServiceControl
2: <constructor>(serviceInstalled)
3: startService(String):boolean
4: stopService(String):boolean
5: <constructor>(serviceInstalled)
ConfigFile
ConfigFile
5.2.1: readFromXMLfile
5.2.1.1:
6.1: writeToXMLfile
ConfigDoc
5.2: ConfigDoc=getConfigDoc(installedService):ConfigDoc
6: setConfigDoc(ConfigDoc):void
instanciation of ConfigGroup, ConfigItem, ConfigAttribute objects when ConfigDoc is instanciated.
ConfigGui
ConfigGui
5.1: setDefaultPathToConfigFile

Figure 3.5: Sequence Diagram of GridSched's ServerGUI.

3.2.2 ClientGUI

Requirements

GridSched's ClientGUI is a graphical user interface for creating and managing jobs on an instance of Titan (see also Section 6.2.2). This service is provided to GridSched's users. Moreover it is possible to initiate the scheduling process and to select a created schedule.
To define a job in the GUI it is necessary to represent the data structure of the job description by GUI elements. A tree view helps navigating the hierarchical structure of the job definition. To support the user defining a job a help window is provided. After defining a job a consistency check is performed.
The user can create, manage and edit only her own jobs. Therefore the authentication of a user is needed.

Architecture

GridSched's ClientGUI consists of five parts and eight classes based on Java Swing technology. Its main concept is the separation of the GUI elements (named `*GUI`) from the functions to control (named `*Control`).

The first part is the main class `GridSchedClient`. It's function is to start and to initialize an instance of the ClientGUI. The second part is the central unit of the ClientGUI. It consists of the classes `ClientGUI` and `ClientControl`. It contains login, user management, job management and schedule management. To communicate with an instance of Titan Remote Method Invocation (RMI) is used. Therefore Titan's communication interface `GUIInterface` is provided. The third part is the application and display manager for the language items "target" and "resource". It consists of the classes `JobGUI` and `JobControl`. When saving the data entered in the GUI all entered data will be checked for consistency. The fourth part realizes the navigation in the structure of the job definition. It consists of the classes `JobTreeGUI` and `JobTreeControl`. The fifth part displays help, version and system messages. It consists of the class `HelpWindow`.

For details of implementation see also Figures 3.6 to 3.14.

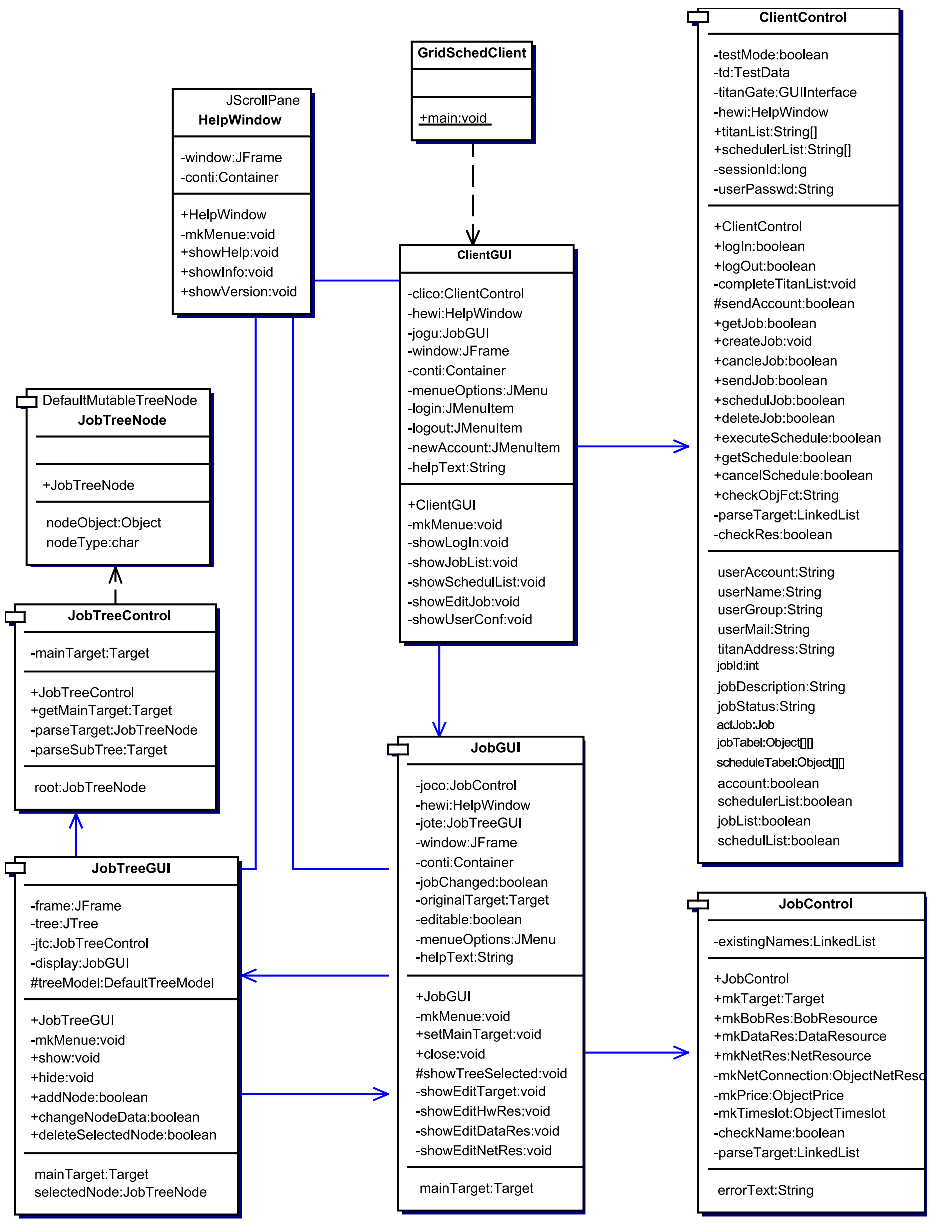


Figure 3.6: UML: ClientGUI.

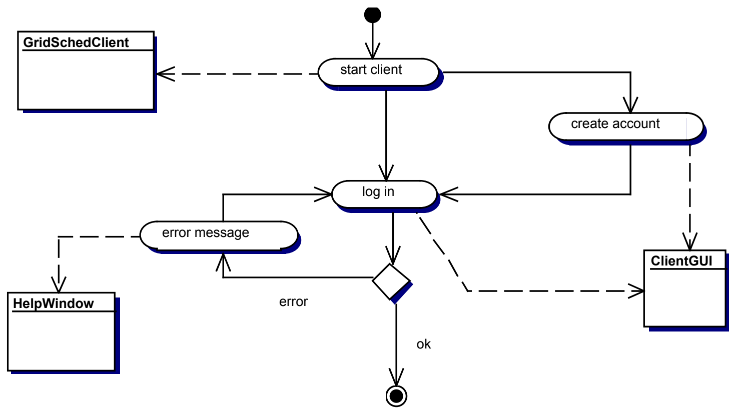


Figure 3.7: UML Activity: ClientGUI - log in.

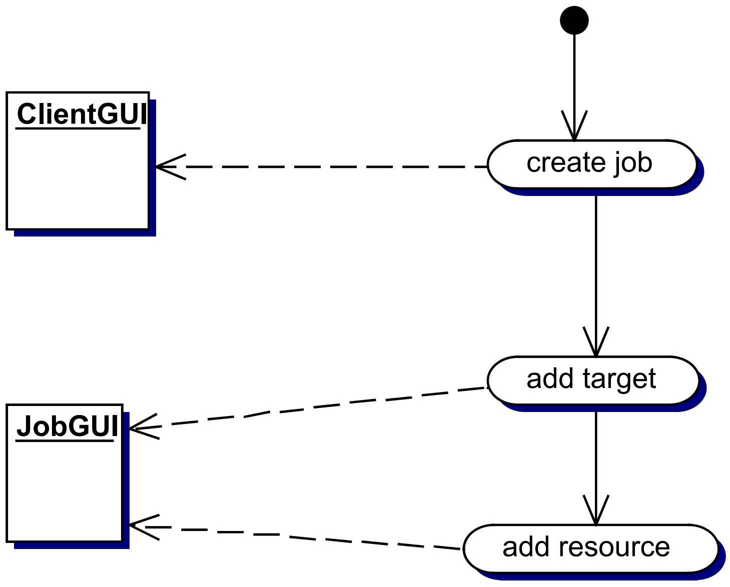


Figure 3.8: UML Activity: ClientGUI - new job.

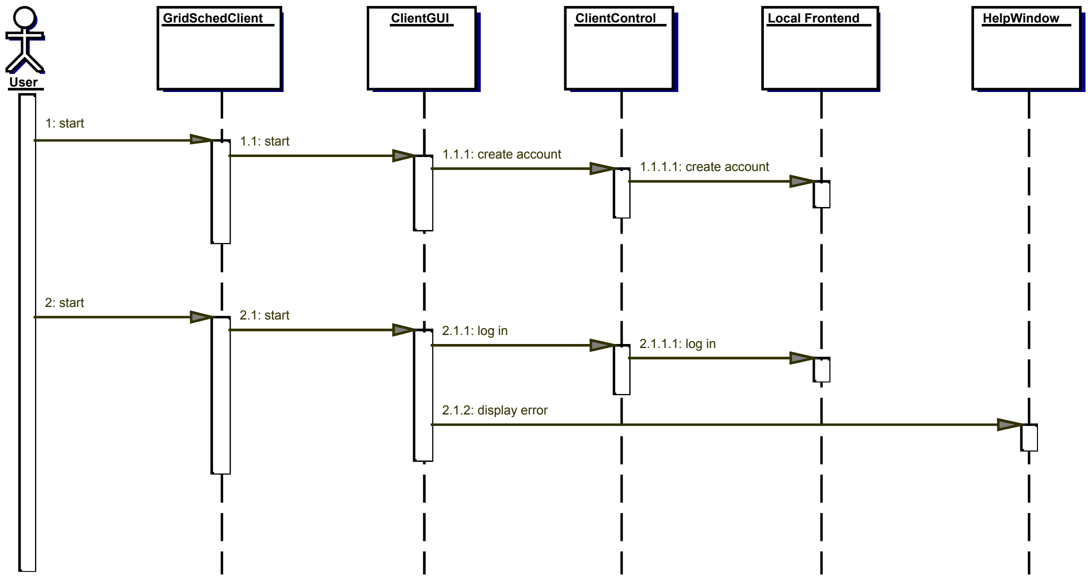


Figure 3.9: UML Sequence: ClientGUI - log in.

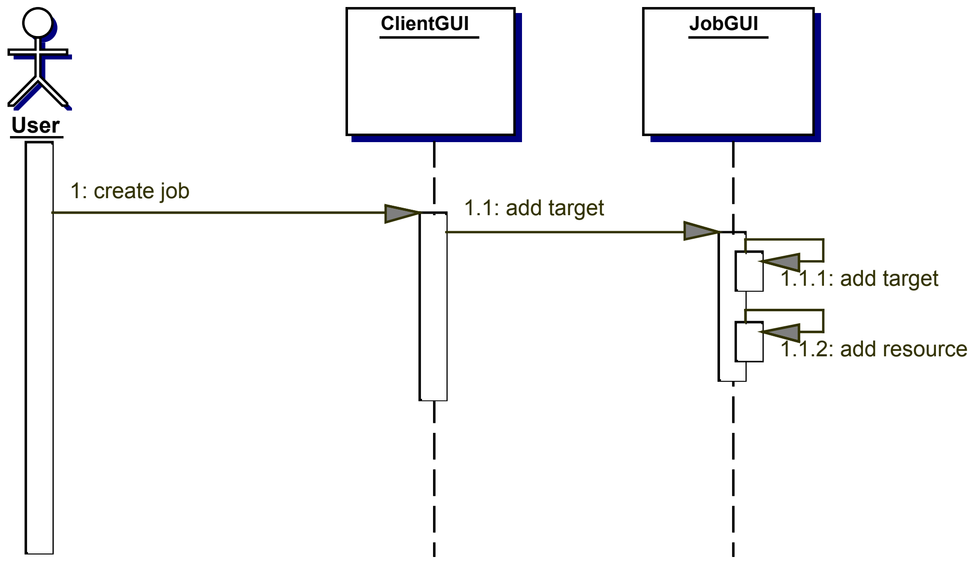


Figure 3.10: UML Sequence: ClientGUI - new job.

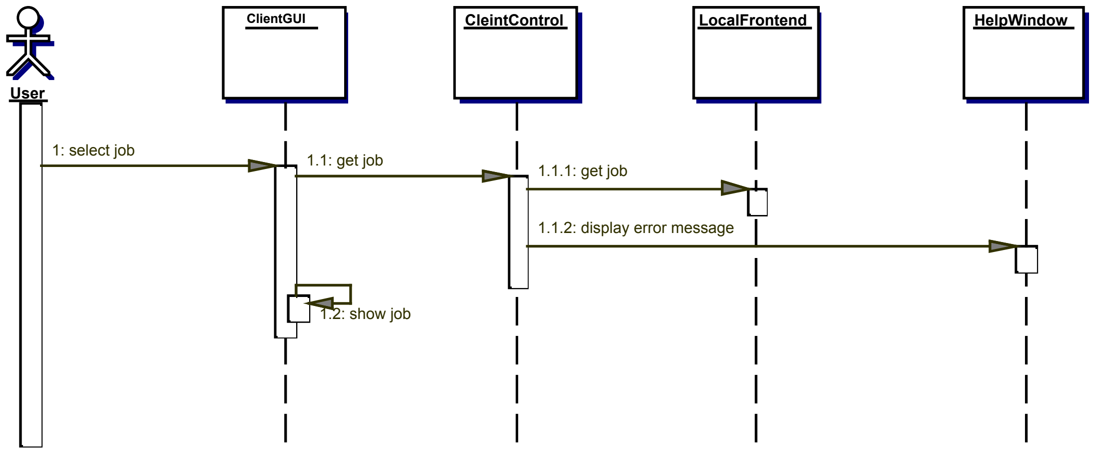


Figure 3.11: UML Sequence: ClientGUI - edit job.

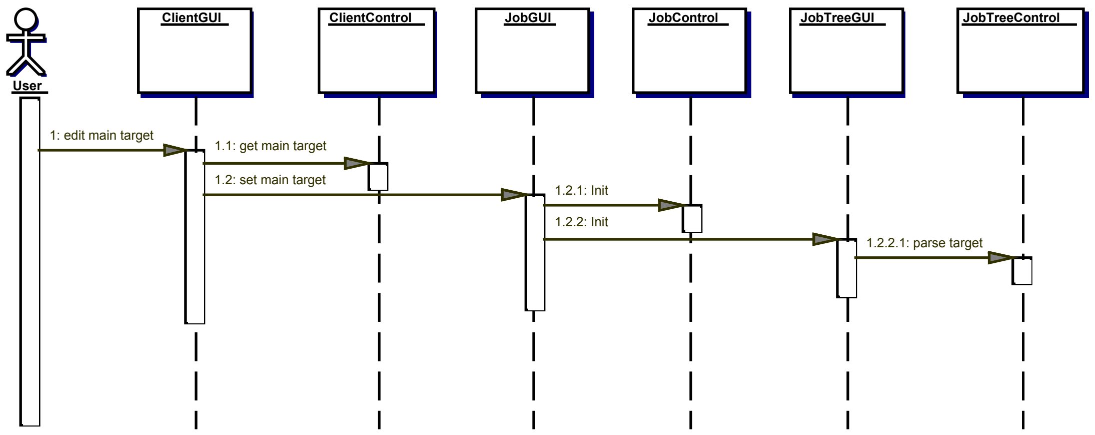


Figure 3.12: UML Sequence: ClientGUI - edit main target.

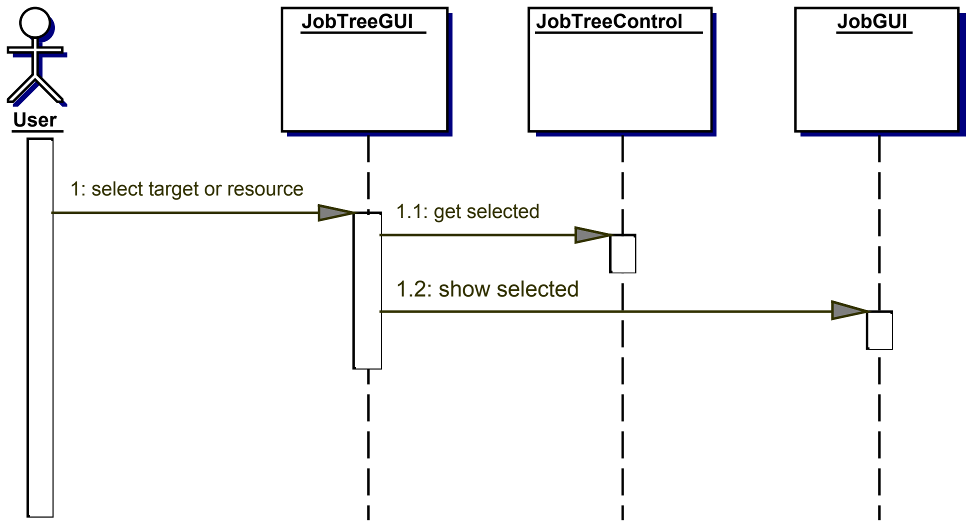


Figure 3.13: UML Sequence: ClientGUI - edit target or resource.

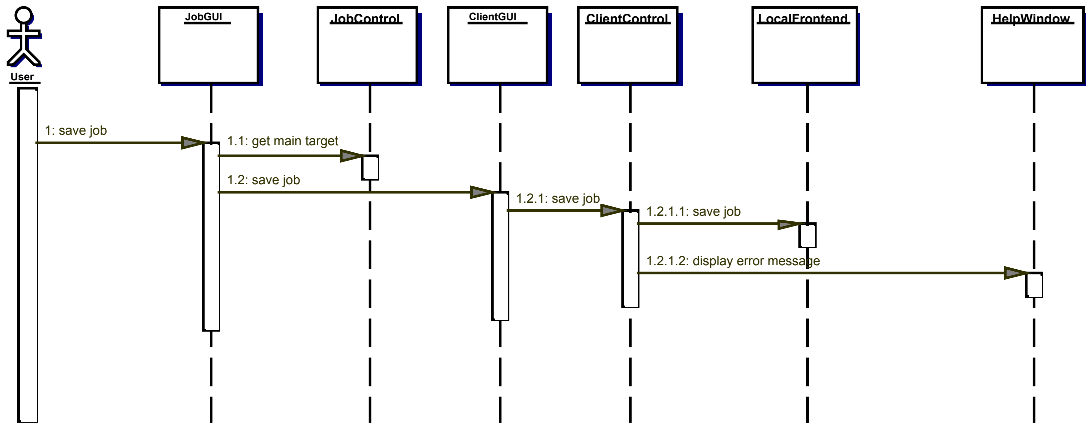


Figure 3.14: UML Sequence: ClientGUI - save job.

3.3 Job Description

To consign a query to GridSched the user has to formulate a job. To do so she can use the provided GUI to commit all data necessary to describe the job to the system. But jobs submitted to the system must be formulated in a way the system can handle them. So to process the jobs created by the means of the GUI, the inputs must be converted to an internal representation after being validated. Therefore a language was created to describe the jobs and their component parts. It consists of data structures which hold all necessary information.

3.3.1 Requirements

What are the requirements of the language? The language should be powerful enough to describe all possible jobs that should be accepted by the system as valid descriptions. It´s main purpose should be to hold the data representing a submitted job and make this data portable. It should be flexible enough to allow future extensions and improvements.

As the language is used throughout all components of the GridSched software, the representations needed by different components differs from case to case. For example the scheduler has to run complex algorithms on a set of resources and therefore represents them in a special format.

Nevertheless, all the implementations have some things in common. To allow local redefinitions of the **language items** without restructuring interoperability, interfaces are used. All language items are defined as interfaces.

3.3.2 Idea and Design

To provide a deeper insight into how jobs look like, this topic will be picked up next.

A job submitted to GridSched mainly contains information about what kind of resources the user needs and in which way she wants them to be used together.

In the most simple case, a computational task handed to the scheduling system includes an executable (for example a shell script), a reference to input data and a specification of the hardware required.

The term **Resource** refers to the description of any kind of hardware, data (file or executable) and network infrastructure.

So first the user has to specify which respectively what kind of resources she wants to use for her job. To do so she can chose from different kinds of resources, e.g. data-, network- or CPU-resources, and specify required and optional attributes. All these resources have counterparts in the language. To be able to state how resources, executable and data are to be used together, a special description called **DoItem** was introduced holding information about the desired interactions between the resources. It also allows different alternative requirements to be expressed. This will be explained in greater detail later.

In GridSched, the objects just mentioned(Resource(s), DoItem) are supposed to form a **Target**. A Target comprises all items that have to be used simultaneously to perform a computation.

Often, a computation results in the creation of data that can be processed further: It's output data can be the input data for a new computation. A **Job** is considered to be a chain of computations, which means that it consists of several Targets. Nevertheless, a Job only including one Target is a valid Job as well.

Moreover a Target can contain an indeterminate number of **Sub-Targets** describing computational tasks which generate data used by their Super-Target and have to be executed before the Super-Target. As said before DoItems can be used to give alternative resource scenarios. This may be useful, but often one possible alternative is preferred by the user.

To enable the scheduler to rate its schedules and select the best one, the user must provide a score function for every job. This function is called **ObjectiveFunction**. It contains information about what kind of resources the user prefers, what she´s willing to pay for the computation and so on.

To get an idea of what items a job contains and how it looks like have a look at Figure 3.15.

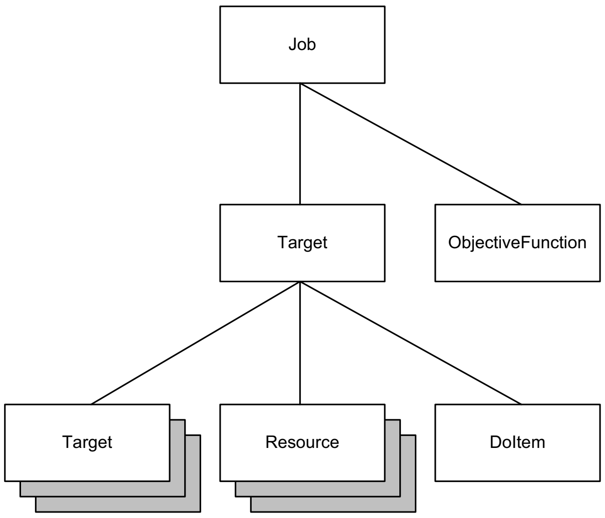


Figure 3.15: Language Item structure.

3.3.3 Implementation

Apart from creating the interfaces for all the items mentioned, it was decided to supply sample implementations for the most important ones. So all components can use the provided implementations of the interfaces or use own, adapted implementations.

The implementations were named : `Object`+name of the implemented interface (e.g. ObjectTarget is the implementation of Target).
To get an overview about the structure, have a look at Figure 3.16 showing all interfaces of the language and their relations.

Subsequently there´ll be a brief explanation of the most important interfaces respectively their sample implementations.

3.3.3.1 LanguageItem

All language interfaces are supposed to extend this one. This serves two different purposes:

1. Make all language objects serializable
2. Force all language classes to implement a function for XML import and export

As you can see there is no need to write an implementation for this particular interface. Nevertheless, the functions mentioned above make it essential for the whole language package.

Considering inheritance, this interface is the root of the inheritance tree.

The function for XML import- and export was not actually implemented as the whole communication model changed during it´s implementation. But for further information have a look at Section 3.3.5.

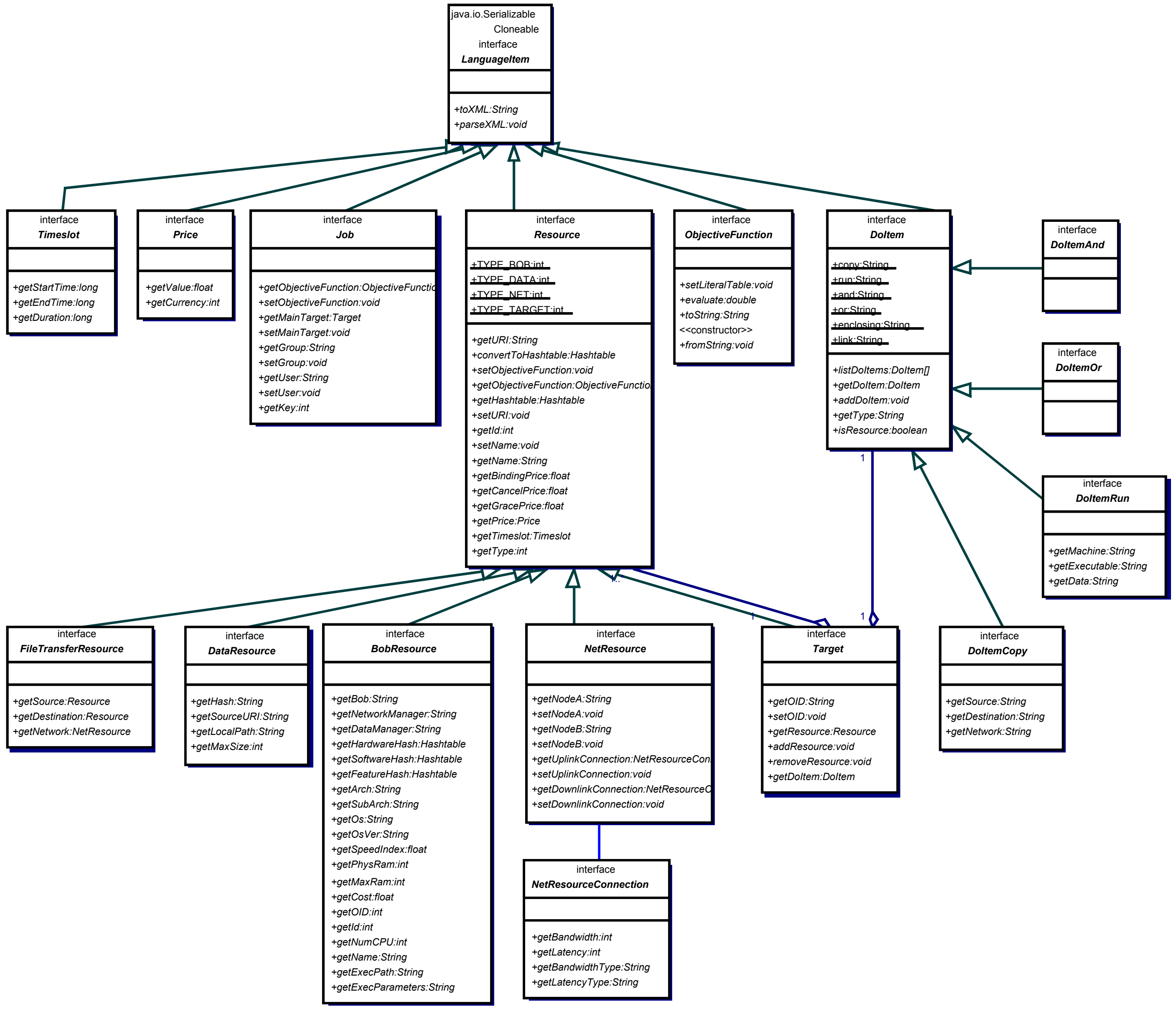


Figure 3.16: Language Interface structure.

3.3.3.2 ObjectiveFunction

The purpose of the ObjectiveFunction is to allow the user to rate a schedule. Therefore it should represent a formula that uses constants and Resource's attributes as variables. For reasons of convenience it should be able to read that formula from a string. The only method it must supply is one that returns a numerical evaluation of the associated formula.

Because the implementation of the ObjectiveFunction is complex and it can be used in all components of GridSched without modification, a sample implementation of this interface is provided. The details of this implementation named `ObjectObjectiveFunction`, will be discussed in the rest of this paragraph.

The content of an ObjectiveFunction is described by a string expressing a mathematical formula. After having created an `ObjectObjectiveFunction-` -object, this string can be passed to it.

The following rules describe the grammar of an `ObjectiveFunction`:

non-terminal symbol	production
`expression`	*term* or *ifThenElse*
`term`	*product* or *unary* or *(expression)* or *literal*
`ifThenElse`	`if` *expression* `then` *expression* `else` *expression*
`product`	*term* `AbstractBinaryOperator` *term*
`unary`	`AbstractUnaryOperator` *term*

Table 3.1: Grammar of the `ObjectiveFunction`.

All the keywords in the table above have corresponding private classes in `ObjectObjectiveFunction`, the sample implementation of the `ObjectiveFunction`-interface. The methods `AbstractUnaryOperator` and `AbstractBinaryOperator` play an important role in extensibility regarding mathematical functionality. New operators can simply be implemented by writing classes that extend the corresponding abstract class.
The +-operator may serve as an example here:

```
private class Plus extends AbstractBinaryOperator {
    public Plus(LinkedList l) {
        super(l,"+");
    }
    public double evaluate() {
        return (arg1.evaluate() + arg2.evaluate());
    }
}
```

Figure 3.17: Code of the +-operator.

Adding new unary and binary operators is very easy. The `if/then/else`-operator should be explained particularly: The `expression` given between `if` and `then` is checked for being non-zero. If that is the case, the `then`-part is evaluated and returned, otherwise it is the `else`-part whose value the statement returns.

The grammar defines Literals that have not been explained yet: A Literal can either be a constant (e.g. 13) or a reference to a Resource's attribute, for example "res1.numCPU".

Internally, an `ObjectiveFunction`-object keeps the resource's attributes in a table that is referred to as **literal table**

On details about how this table is initialized, please see Figure 3.3.3.3.

3.3.3.3 Job

To understand the structure of the interface `Job`, it is important to recall the language objects' structure explained in Section 3.3.2. Resources which should/must be reserved simultaneously are collected to Targets to be able to process the computation's output. A Job was defined as a chain of computations. So a Job contains one root target that may contain subordinate Targets.

Another important point is that a Job contains an ObjectiveFunction. As explained in the previous paragraph, an ObjectiveFunction references Resources' attributes. Therefore it uses the literal table. It is the task of a `Job's` `getObjectiveFunction`-method to initialize the `ObjectiveFunction's` literal table before returning the `ObjectObjectiveFunction`-object. This works as follows: The `Job` queries its `Target` to return all attributes in a hashtable. The `Target` calls all subordinate `Targets` and all directly associated `Resources` for their attribute hashtable. Figure 3.18 illustrates this process.

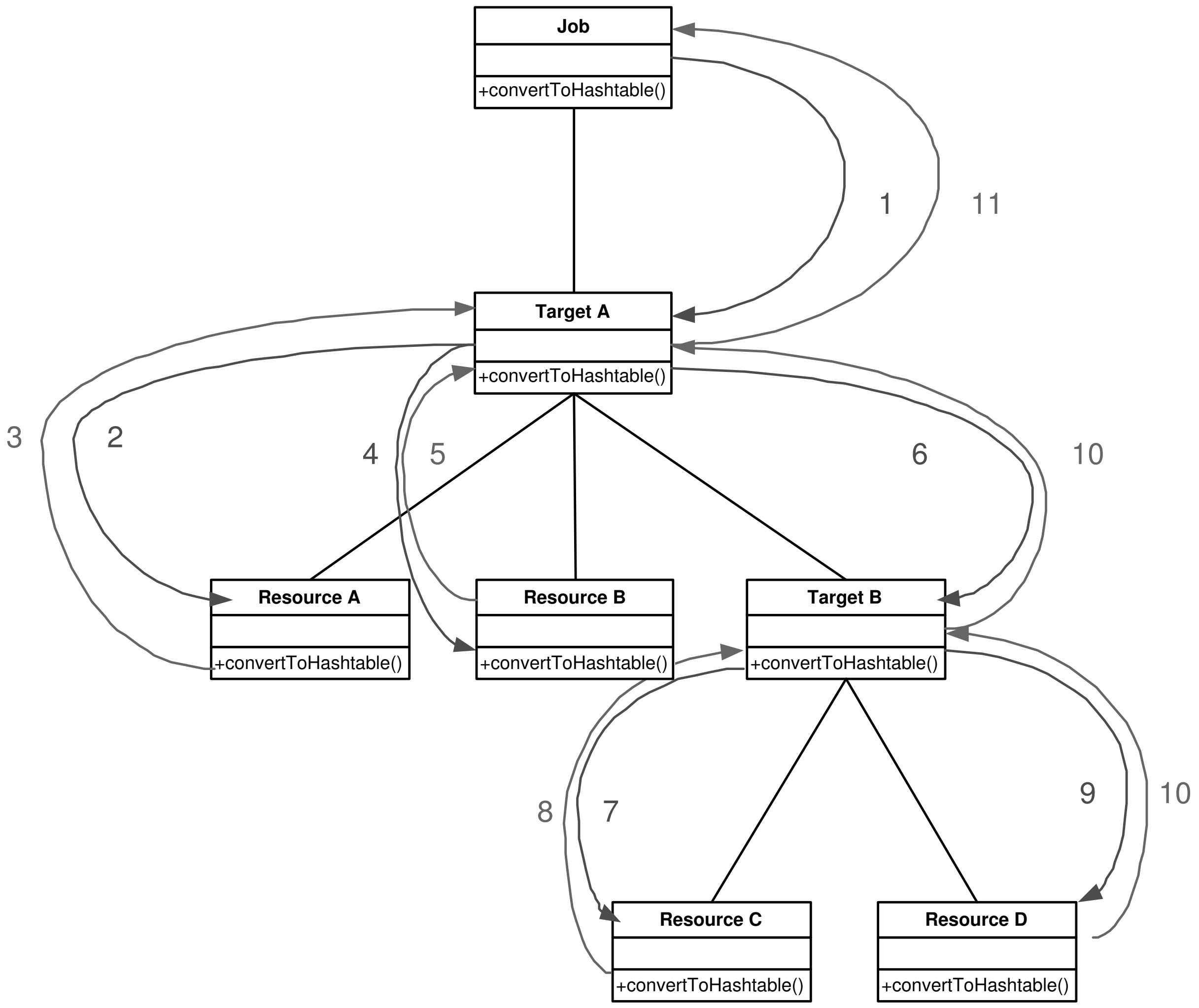


Figure 3.18: Sequence of convertToHashtable.

3.3.3.4 ObjectBobResource

This class represents resources that belong to a `Bob`, which means hardware resources. `ObjectBobResource` has a lot of attributes that will be explained later. How does the `job`-object gather all the attributes from all its resources? See Figure 3.18 to get the answer.

What attributes does an `ObjectBobResource` have? In general, there are two types of attributes, **fixed** and **variable** ones. The fixed ones occur in every `ObjectBobResource`-object and are therefore implemented as attributes. The variable ones are stored in a hashtable. Their purpose is to describe random qualities of a resource. For example, if the user asks for the software *XMMS*, version 1.2.8 to be installed on the system. Two problems occur regarding this kind of request:

- There are two many different products around to have a static attribute for each of them
- New products appear every day, so you must be able to request them even if the Grid software does not know of their existence

To solve this problem the **feature hash** is introduced, which carries pairs of the form (*feature name*, *value*). *Feature name* typically states the name of a software product whereas *value* states its version. With this mechanism it is possible to describe the features of a resource in a very flexible way.

For a detailed description of each of the static attributes please have a look into the Javadocs.

3.3.3.5 ObjectDataResource

The class ObjectDataResource mainly consists of attributes which are necessary to describe a wanted Data Resource and of the corresponding getters and setters.

3.3.3.6 ObjectNetResource

For the sample implementation the class `ObjectNetResource` implementing the interface `NetResource`. This class consists primarily of attributes which are necessary to describe a wanted network connection containing attributes like endpoint a, endpoint b, needed bandwidth and latency, maximal price. With this implementation it´s possible to specify bandwidth and latency for both uplink and downlink connection.

3.3.3.7 DoItem

Every target consisting of different resources has an associated DoItem which specifies what to do with the resources. For example it could be possible to execute **program A**(resource 1) on **CPU B**(resource 2) using **data C**(resource 3). So DoItems actually describe how sets of resource defined in a target should be used together. As the DoItem is a recursive structure, a DoItem has a list of other DoItems. This particularity results from the fact that there are different kinds of DoItems. For the sample implementation the classes **ObjectDoItemRun**, **ObjectDoItemCopy**, **ObjectDoItemAnd** and **ObjectDoItemOr** were implemented, all implementing the interface **DoItem**. Using these classes jobs consisting of either executing a program on a specified kind of computer using a specified Data Resource or copying some kind of data between specified locations can be phrased. Of course the operations can be combined. It is also possible to create queries containing logically concatenated DoItems using the classes **ObjectDoItemAnd** or **ObjectDoItemOr**.

To pickup up the preceding example:
If a target containing the CPU resource **CPU** and the Data Resources **data1** and **data2**, data1 being executable data, is specified and it should be expressed that **data1** should be executed on an CPU of kind **CPU** using data of kind **data2**, this target would have a DoItem of type **ObjectDoItemRun**.

Of course the user should not be enforced to create the nested objects of the different DoItem classes to phrase her query. The user specifies her query as a string using the provided GUI. Internally an object of the class ObjectDoItem is created and the method of this class to parse the query is invoked. If the query complies with the given syntax, the different nested objects are created to transform the query to a form understandable by the system.

non-terminal symbol	production
`statement`	*RUN* or *highLevelStatement*
`highLevelStatement`	*AND(statement)* or *OR(statement)* or *RUN*
`RUN`	*RUN* `EXE` *ON* `CPU` *USING* `data`
`EXE`	String specifying an executable
`CPU`	String specifying a CPU
`data`	String specifying a dataset

Table 3.2: Grammar for the DoItems.

In the sample implementation the following grammar is used to regiment the format of the textual queries:
So all queries respecting this grammar are valid and can be processed by the system. There are two more things to mention about the textual queries. As can be seen in the presentation of the grammar, strings can be used to specify CPUs, data sets and executables. These strings can either specify just types of resources or concrete resources like "use the CPU of the computer with IP address 10.30.135.11" . Furthermore the strings are regimented, too. You can only use strings representing resources that are part of the surrounding target. Those strings are held in a hash table.

3.3.3.8 Helper classes

There are some classes that are used in different parts of the software but are not part of its core functionality. Their purpose is to prevent inconsistencies and redundancy.

The first helper class to mention is the class `ObjectTimeslot`. Often the problem of having to represent a time period may occur, which is the combination of a start time, an end time and a duration. This class provides a convenient notation for this.

`Price` was made a separate class because this encapsulation allows to abstract from concrete currencies. While internally the software deals with numbers that do not represent a particular currency, a feature that allows the user to determine an exchange ratio by setting a variable, is imaginable. This exchange ratio would only appear in the `price`-class.

3.3.4 Example

To illustrate the structure of the language once again an example will be presented . A certain job and its representation in the language will be shown next. To alleviate understanding have a look at Figure 3.19.

The figure elucidates the hierarchical structure of the language.

The Job-object is the root of the tree. It contains all other items, basically it´s just a passive container having no active functions. As said before a job contains at least one target and an objective function. So does the sample job. Target A is composed of 6 resources, one being the result of another computation, represented by Target B. Except of Target B representing the data resulting from a former computation the other resources do not depict concrete resources(e.g. program xy on computer 10.30.135.202) but templates for resource types. They isolate the pool of resources the scheduler can pick to satisfy the query. To give a possibility to rate the schedule the objective function is consigned to the schedule. It is necessary for the scheduler to pick the resources which fit the query best and to improve its results.
The DoItem belonging to Target A prescribes how the resources should be used together, basically data should be copied to a specified kind of CPU and be executed there. Of course Target B must be executed before Target A, as its result should be used in Target A.

Creating the query the user just fills the fields she´s interested in. If she wants a variable amount,

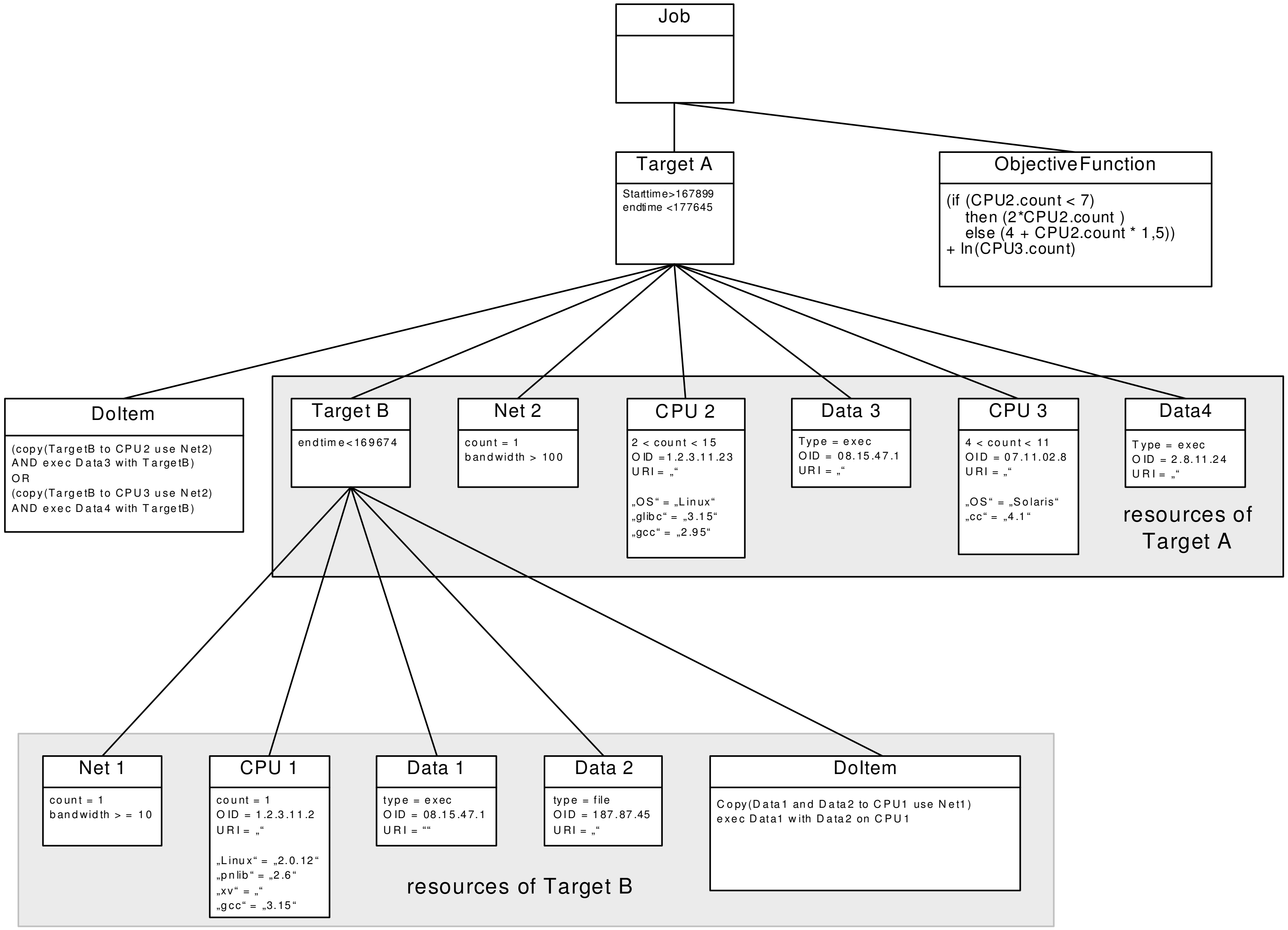


Figure 3.19: Example Job.

maybe between 2 and 15, of a special kind of CPU, the CPU belonging to a computer with Linux installed, she would just fill the fields for the type of CPU, the OS, and the amount of CPUs.
After scheduling finished successfully, the resource-objects represent concrete resources and all fields available from the resources are filled.

3.3.5 Conclusion

The language changed just slightly since the first term. It was intended to implement functionality to export the Java objects to XML and vice versa. The interfaces reflect those intentions as they contain methods for these conversions. It was finally decided not to implement this functionality in the sample classes as our plans to change the communication were not implemented. The decision was made that SOAP, which definitely would have needed the XML im- and export, would not work properly with the system. So RMI is still used for message passing and there´s no need for XML.

3.4 Super Scheduler

Roles in Grid Scheduling may be divided in at least two classes: those who provide resources and those who employ resources to run a job. To provide access to their resources, the providers may set up local schedulers that manage their utilization. To select those resources that would be employed in a job, another scheduler must negotiate with the local schedulers of the resources that are participating in Grid Scheduling to find those among them who are suitable and available. This kind of scheduler is called Super Scheduler.

In GridSched a component is needed that allows users to submit, manage, schedule and execute jobs for the Grid. That component was named 'Titan'.

3.4.0.1 Requirements

- **Stand alone service** Titan should be a service which would be available to multiple users and which can be accessed remotely and in parallel to allow better utilization of information which were gathered about the Grid.
- **Convenient input and management of jobs** Titan should feature a user interface that allows easy assembly and specification of the resources that a user wants to employ in a job. This includes the possibility of conveniently creating objective functions and DoItems.
- **Flexible configuration of the scheduling algorithm** As not all scheduling concepts and strategies might be suited for all kinds of jobs and Grid situations, Titan should provide means to employ different schedulers that a user can choose from to process her job.

3.4.0.2 Design

Core The central components of Titan are formed by a JobManager. The JobManager keeps track of Jobs that where submitted by users. Jobs are stored in wrapper classes named TitanJobs. These wrapper classes hold additional information on who submitted the job, whether or not it was successfully scheduled and what kind of scheduler was assigned to handle the job.

Frontend Over the course of the project, two frontends were created.

- The first version centers around the UserProxy class which provides a rather small set of operations to a web based interface. The development of the Web Frontend was stopped after the first term. Please see [13] for details.
- The second frontend exports access to a wider array of operations via Java RMI. These operations are used by the client GUI package explained in (Section: 3.2.2). The implementing class is named LocalFrontend.

Along with that frontend, several new features were added to Titan:

- **Account Management** An AccountManager keeps track of TitanAccounts which hold information about a users contact information, access password and of the jobs the user has submitted so far.
- **Session Management** In order to perform any action using the LocalFrontend, a user must first identify himself using the username and password stored by the account manager. The user is then issued a sessionId which must be sent along with all entailing method invocations.

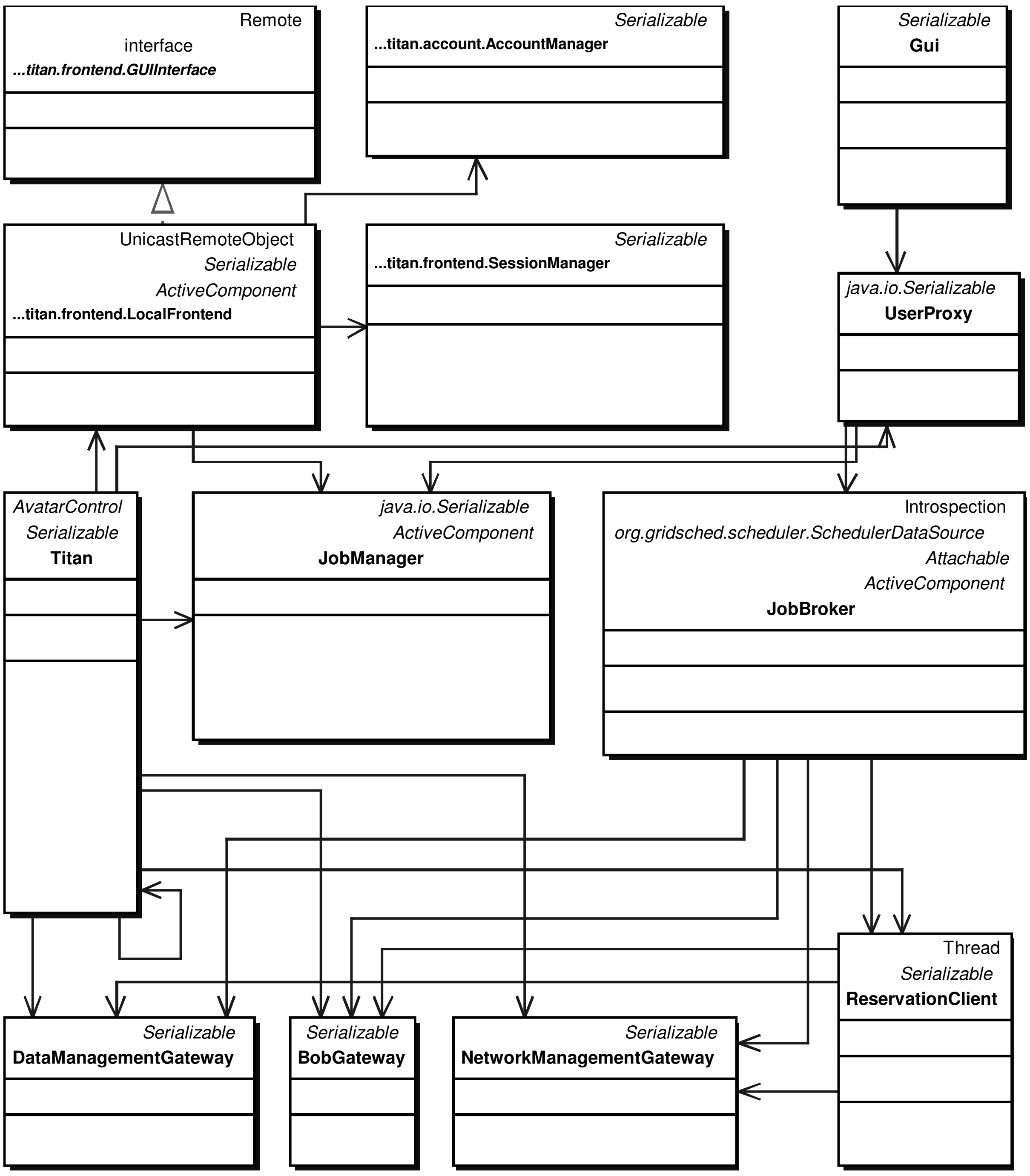


Figure 3.20: Classes of Titan.

- **Access Permission Management** A set of actions was created for use with GridSched's own Policy Manager (Section: 3.9). In order to perform an operation on the LocalFrontend, the PolicyManager must grant access for the corresponding action. For a description of the possible operations, please refer to the Interface
`org.gridsched.titan.frontend.GUIInterface`.

The validity of the SessionId as well as the permission to perform the desired action is verified for every method invocation of the local frontend to provide some degree of security.

Backend Backend classes provide access to the Global Grid. They form something like a 'Grid abstraction layer where any kind of resource can be interacted with in a unified way by means of the protocol mentioned in (Section: 3.1.1.1). For each type of resource a corresponding gateway was created that implements the set of common access methods. The JobBroker class provides unified access to any existing gateways and serves as SchedulerDataSource for schedulers which want to access the Grid. Reservations are handled by the ReservationClient class which allows reserving the resources of a previously scheduled job by means of a simple negotiation scheme. (See Paragraph: 3.1.2 at page 24)

3.4.1 Modular Scheduling Strategy Subcomponents

In this section the different Schedulers implemented for 'Titan' are presented.

The Scheduling Problem Technically, scheduling, as it is understood in the scope of the project, is the process of creating a plan when, and in which order and how to perform certain actions. In GridSched, it is tried to reserve and use certain kinds of resources at given times.

Additionally it is necessary that those resources are selected in such a way as that the objective value of this schedule is optimal. The objective value is produced by an objective function, which takes the resources' properties and uses them as input for a complex arithmetic expression. Evaluation of that expression produces the objective value.

A schedule can be interpreted as a complicated system of equations whose variables are the resources' properties. For each resource there is one equation. This equation is met if its variables are selected in such a way that there exists a resource on the Grid whose properties match those variables' values. These equations can be solved by querying the Grid for offers on that resource. A corresponding local-scheduler will inspect the variables already determined and will try to select the rest of them accordingly. The result is a selection of variables which meet this single equation. There is one equation that describes how resources are to be employed in conjunction. That equation can be created by inspecting the job's structure and its DoItems. Effectively, DoItems can be translated in an equation. The equation is met, once all resources have been selected in such a way that all dependencies between resources are met. Consider the following simple example:

Consider a job that employs two separate computers to perform some kind of distributed computation. To do this, it is essential that both computers run their part of that computation at the same time, because the two will eventually have to exchange data from time to time. So there is a dependency between the timeslot that the first computer should be reserved for and the timeslot of the second computer. Furthermore it must be guaranteed that communication between the two computers will be sufficiently fast. To achieve this it is necessary to reserve a network connection between those two computers that provides enough bandwidth for the data that are expected to be exchanged and that features a sufficiently low latency. The parameters of this network connection depend on those of the computers that are employed in the job. Obviously that connection must be reserved for the same timeslot as those of the computers employed. Additionally the addresses of the sites that the network connection is meant to connect must be provided. These addresses would obviously be the same as the addresses of the two computers. So there is a relation between the computer's site's addresses and the network connection's start- and endpoint parameters. The objective function is the final equation that has to be maximized. Its value can be defined as negative infinity as long as one of the other equations are not met.

In the following a brief overview about the methods considered for solving this problem is provided.

- **Linear Optimization** Maximization of a functions' value seems to be a linear optimization problem. Unfortunately this strategy requires that all variables can somehow be converted into non-discrete numbers. Furthermore all equations have to be linear. None of both is the case.
- **Dynamic Programming** At first it seems like selecting one resource after another in such a way that for each resource it is picked the one which is suited the most for participating in the job might be a good idea. Unfortunately, even if there was a function which decides which resource-option is the best, there is no guarantee that selecting those resources will result in a good objective value. Consider the following example:

  Suppose that there was a simple objective function which returns an optimal value if all resources' prices are minimal. It looks like a good strategy to pick the cheapest option available for each resource. Unfortunately, due to other dependencies it may occur that by selecting a cheap option for one resource there will be no cheap options available for other resources. So Bellman's equation, which is essential for dynamic programming, does not hold.
- **Branch and Bound** If there was one valid solution to the system of equations it would be possible to enforce or prohibit some values of some variables and see if a better solution comes about when that modified equation is re-solved. Since it is possible to find a solution to the system of equations the objective value of that solution can be used as a lower bound. The problem is finding an upper bound for the remaining problem. It is possible to remove all equations but the objective function itself and take that as a relaxation of the problem. The task would then be to compute maximum value possible under the current constraints of enforced or prohibited values. But, since the objective function can be of whichever form it would be necessary to resort to genetic algorithms to do a black-box optimization of that function. This approach did at least seem feasible and was attempted during the implementation of SimpleSched.

A common strategy Under the circumstances presented, most of the traditional approaches of computer science seem fated to fail. The only way that appears to be left is to resort to heuristic approaches like genetic algorithms. For that reason, all schedulers implemented for GridSched up to today work in the following way:

```
Compute a single valid schedule;
Store that schedule;
While the Algorithm should continue (
    Select a previously stored schedule;
    Modify that schedule;
    Compute a new valid schedule that is based on the modified one;
    Store that new Schedule;
)
```

Usually, genetic algorithms can produce useful results only if they perform enough iterations, so the main concern was to find efficient ways to compute a valid schedule.

In order to make the use of different schedulers as transparent as possible a common interface was created that defines the basic operations that each scheduler must support. Some of these are:

- `assignJob(Job theJob)` An existing instance of a scheduler is assigned a job for which it should produce schedules.
- `assignDataSource(SchedulerDataSource theSource)` The scheduler is assigned a reference to a SchedulerDataSource, which serves as a source of information on existing Grid Resources. Usually this would be an instance of Titan's JobBroker class. (See Section: 3.4.0.2)
- `startScheduling()` This will commence the scheduling activities.
- `getSchedules()` At any time after invocation of `startScheduling()`, calling this method can be used to get a most recent list of schedules produced which comes in ascending order, sorted by the objective value of the schedules.

```
/**
 * Start with a single empty schedule.
 */
Schedule = new Schedule();

do {
    /**
     * Find resources on the Grid which are suitable
     * for accomplishing the job.
     */
    Resources = discover(Job);

    /**
     * Contact those resources to find out which of
     * them are available at the given time.
     */
    Timeslots = check_availability(Resources);

    /**
     * Select Resources in a predefined-defined order.
     */
    for( all resources needed for schedule ){
        Schedule.add( get_offer(Timeslots) );
    }

    /**
     * If successful, add the newly created Schedule
     * to a Set of Schedules.
     */
    Schedules.add( Schedule );

    /**
     * Create a variant of the best Schedule and
     * re-compute it, hoping that it will become
     * even better.
     */
    Schedule = make_variant( Schedules.get_best_schedule() );

} while( scheduler_should_continue());
```

Figure 3.21: Algorithmic description of SimpleSched.

Over the course of the project, two schedulers where designed. They were named SimpleSched and NuSched.

3.4.1.1 SimpleSched

SimpleSched was created for the prototype of GridSched. The goal was to write a scheduler that would find a solution at all, so the objective was to keep it simple.

To find a good schedule for the resources defined in a given job, the SimpleSched would perform the steps as seen in Figure: 3.21

Design This paragraph will elaborate on the following areas of interest:

1. Representing a schedule
2. Finding resources on the Grid and getting timeslots for them
3. Selecting among the resources available
4. Creating variants of a complete schedule

Finally there will be a vague estimation of the algorithms complexity and performance, along with a summary of SimpleSched's features.

1. **Representing a schedule** Each job that needs scheduling is assigned an instance of the scheduler that would run in an own thread. Once started the scheduler would convert the job into an internal data structure, the ProblemSpace. There (See Figure: 3.22) any resource or target is represented as a Problem.

2. **Finding resources on the Grid and getting time slots for them** The task of the scheduler is to find a solution for each problem recursively until the root problem of the ProblemSpace is solved. To do that, the scheduler relies on a set of resources it queries from the Grid. These resources are organized in a data structure called OptionsPool. It reflects the options the scheduler has to solve a problem. This set is made of instances of non-dynamic resources that where found on the Grid. The process of finding these resources is called population. The options pool is populated once at initialization time. It looks for instances of all non-dynamic resources it finds in the current ProblemSpace. For each instance retrieved, it immediately requests time slots and stores them as options. Once the root problem of a ProblemSpace has been solved a valid schedule has been computed. That Solution is added to the SolutionSpace. (See Figure: 3.26)

3. **Selecting among the resources available** The order, in which problems need to be solved on the one hand comes from the structure defined by the hierarchy of targets as found in the job. On the other hand, the do-items found in each target introduce a second, more detailed level of dependency. They describe in what time and logical order resources need to be allocated. Analyzing a given job, the ProblemSpace organizes the resources in a dependency tree. The LowerBoundSolver takes a ProblemSpace and traverses its dependency tree. It uses backtracking to try out possible combinations of options for different resources until a valid solution for the root-problem is found or the algorithm runs out of options for the root problem.

   Resources must meet certain constraints to be valid. Such constraints arise from the dependency structure within the job or are provided by the user. Typical constraints are time slots or budget. Usually the user sets a limit on the price she wants to pay for the execution of her job. Normally she would also enter a time window within which the job should take place. The LowerBoundSolver must take into account these constraints when selecting resources. Constraints change as more and more problems of the ProblemSpace are solved, budget decreases, time slots for certain resources depend on the availability of others. These changes are propagated as the LowerBoundSolver traverses the ProblemSpace.

   The LowerBoundSolver has access to the ProblemSpace and to the corresponding OptionsPool. To create a Schedule it performs the steps as seen in Figure: 3.24 and Figure: 3.25 .

4. **Creating variants of a complete schedule** Once a schedule has been computed, it is stored in a SolutionSpace. Its task is to keep track of the solutions already produced and to create new problems to be solved. This is called branching and is achieved by removing selected options from the options pool of the current problem. This will prevent the LowerBoundSolver from reproducing the last solution and force it to create a new one. (See Figure: 3.26) Initially, the SolutionSpace contains nothing but one node. The root-SolutionSpaceNode. When the SolutionSpace is called to produce solutions, it goes through the following loop, until a stop-criterion is met.

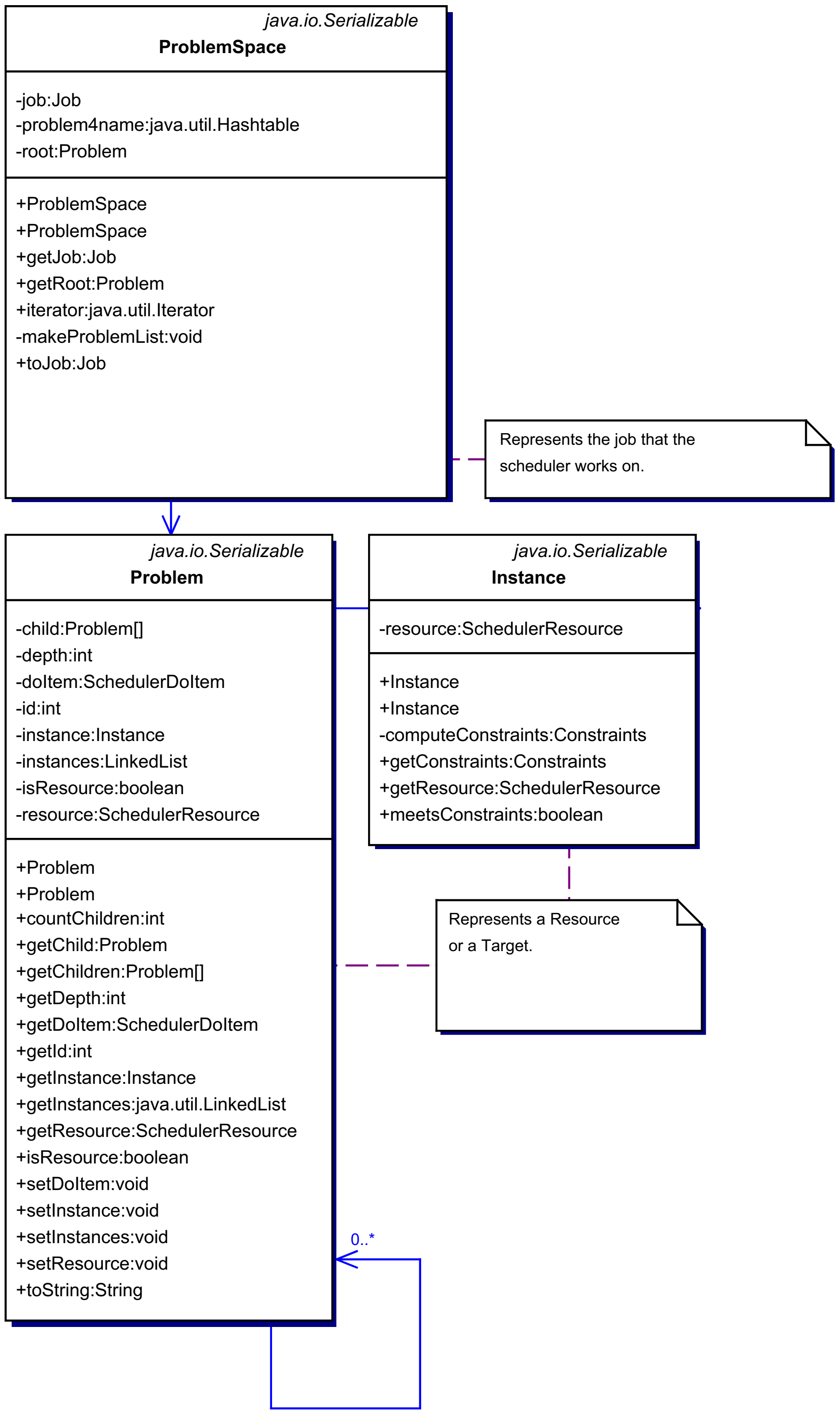


Figure 3.22: Classes related to the ProblemSpace classes of SimpleSched.

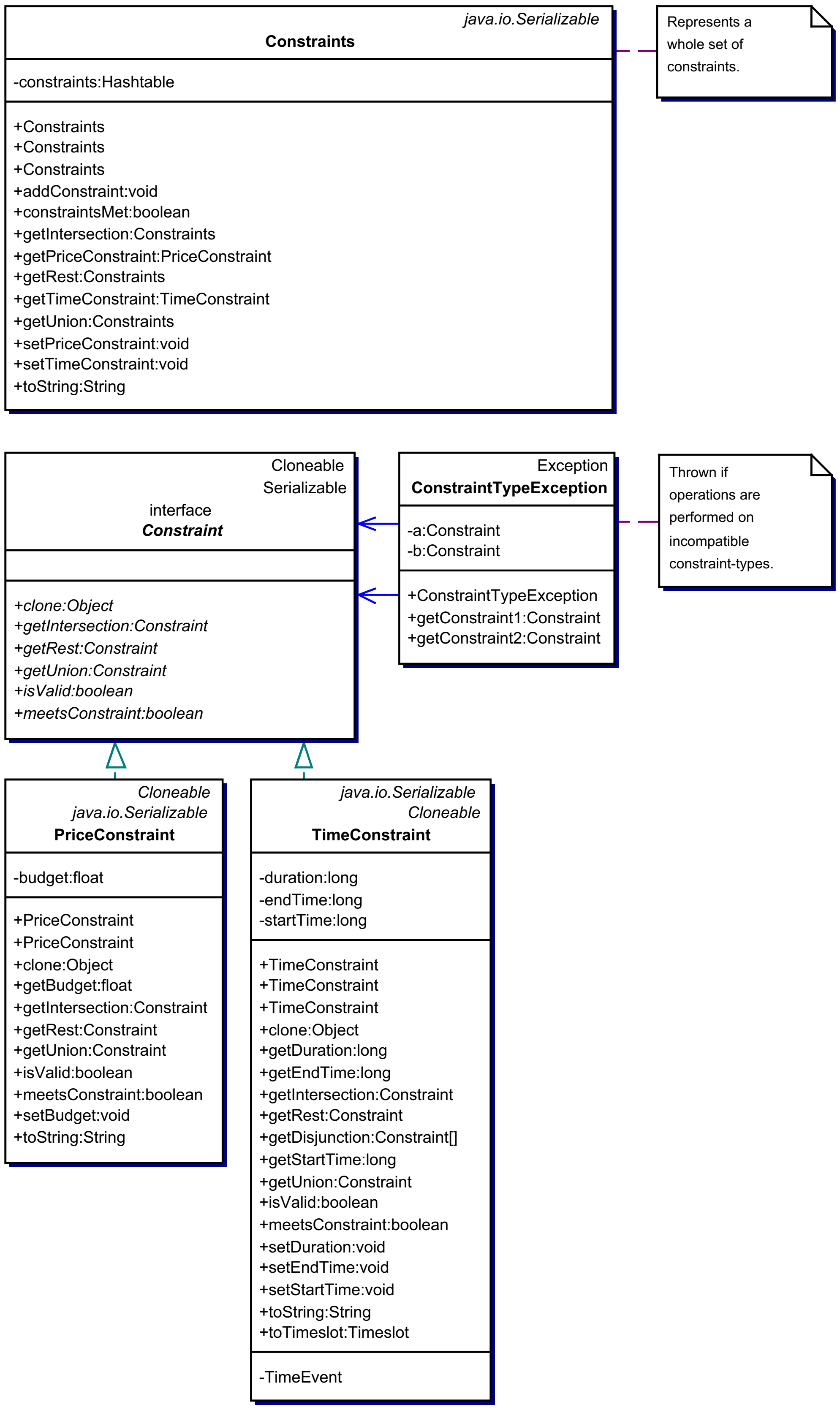


Figure 3.23: Classes related to the implementation of Constraints within SimpleSched.

```
constraints = get the job's main-target's constraints;
doItem = get the job's main-target's DoItem;
BottomUpCompute(doItem, constraints);
forward an exception if the call fails;
compute and return the objective value for that solution;

//recursive method
BottomUpCompute(doItem, constraints){
    if(doItem reflects a resource){
        if(the resource is non-dynamic){
            get an option for that resource
                from the OptionsPool;
            throw an exception if there is no such option;
        }
        else {
            query the SchedulerDataSource for
                an offer for that dynamic resource
                that fits the current constraints;
            throw an exception if there is no such offer;
        }
        set the result as a selected instance
            for the resource;
        return;
        }
    else{
    for(all child-doItems doItem'){
        BottomUpCompute(doItem', constraints);
        if(the call returned successfully){
            update the constraints;
        }
        else {
            go back to the last child-DoItem
                that was processed successfully;
            throw an exception if there is
                no such child-DoItem left;
            invalidate that DoItem's selected
                resource instance;
            restore that DoItem's constraints;
        }
    }
}
```

Figure 3.24: Pseudocode representation of the LowerBoundSolver's strategy for creating a valid schedule.

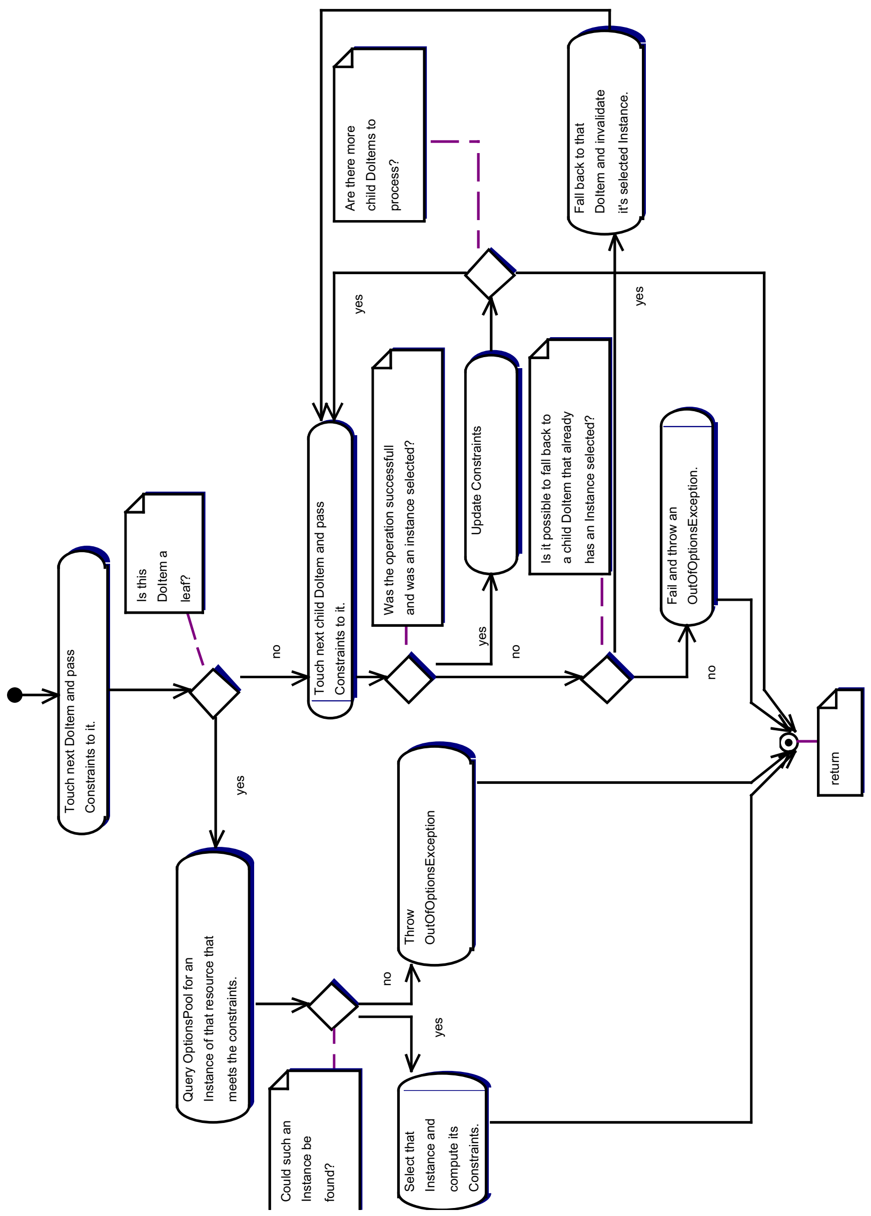

Touch next DoItem and pass Constraints to it.
Is this DoItem a leaf?
no
yes
Touch next child DoItem and pass Constraints to it.
Was the operation successfull and was an instance selected?
yes
Update Constraints
Are there more child DoItems to process?
yes
no
Is it possible to fall back to a child DoItem that already has an Instance selected?
yes
Fall back to that DoItem and invalidate it's selected Instance.
no
Fail and throw an OutOfOptionsException.
return
Query OptionsPool for an Instance of that resource that meets the constraints.
Could such an Instance be found?
no
Throw OutOfOptionsException
yes
Select that Instance and compute its Constraints.

Figure 3.25: Activity diagram of the LowerBoundSolver's strategy for creating a valid schedule.

```
(1)Take an unsolved SolutionSpaceNode and
    have a LowerBoundSolver compute a solution for it.
(2)If that succeeds, call branch on that node,
    creating new unsolved nodes.
```

Solutions are converted into jobs and stored in a list of schedules. Elements of that list are sorted by their objective-value as returned by the associated objective function. The loop continues as long as there are unsolved nodes left or as long as none of the stop-criteria is met. Currently the only stop-criterion is the number of iterations the loop does. It is met once the loop has been repeated a certain number of times. Another criterion may be met once the best objective-value has not been improved by a certain percentage for a given number of cycles.

The implementation allows to retrieve a schedule from the scheduler even when the algorithm has not completely finished yet. It allows the user to follow the improvements the scheduler achieves

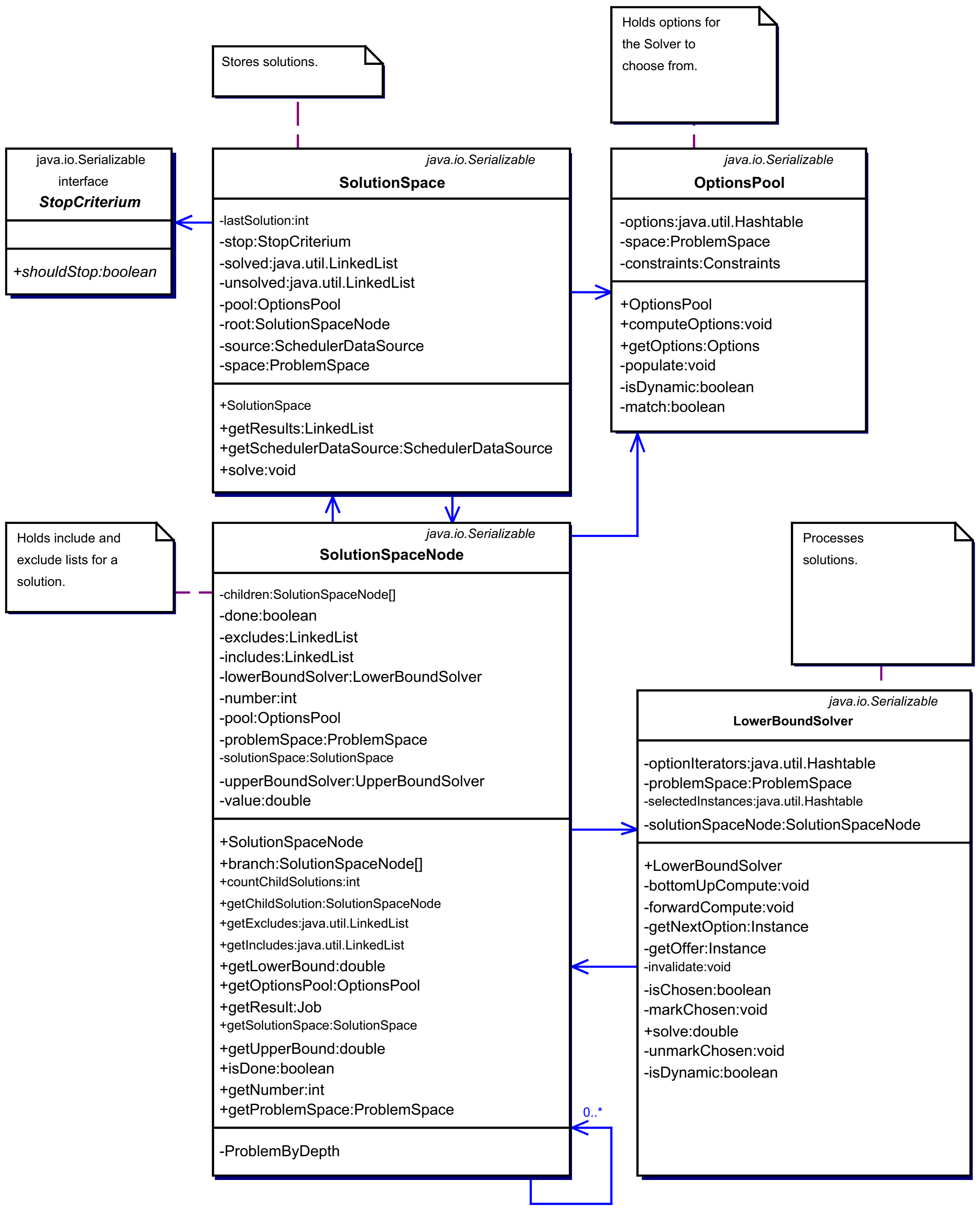


Figure 3.26: Classes related to the SolutionSpace sub-module of SimpleSched.

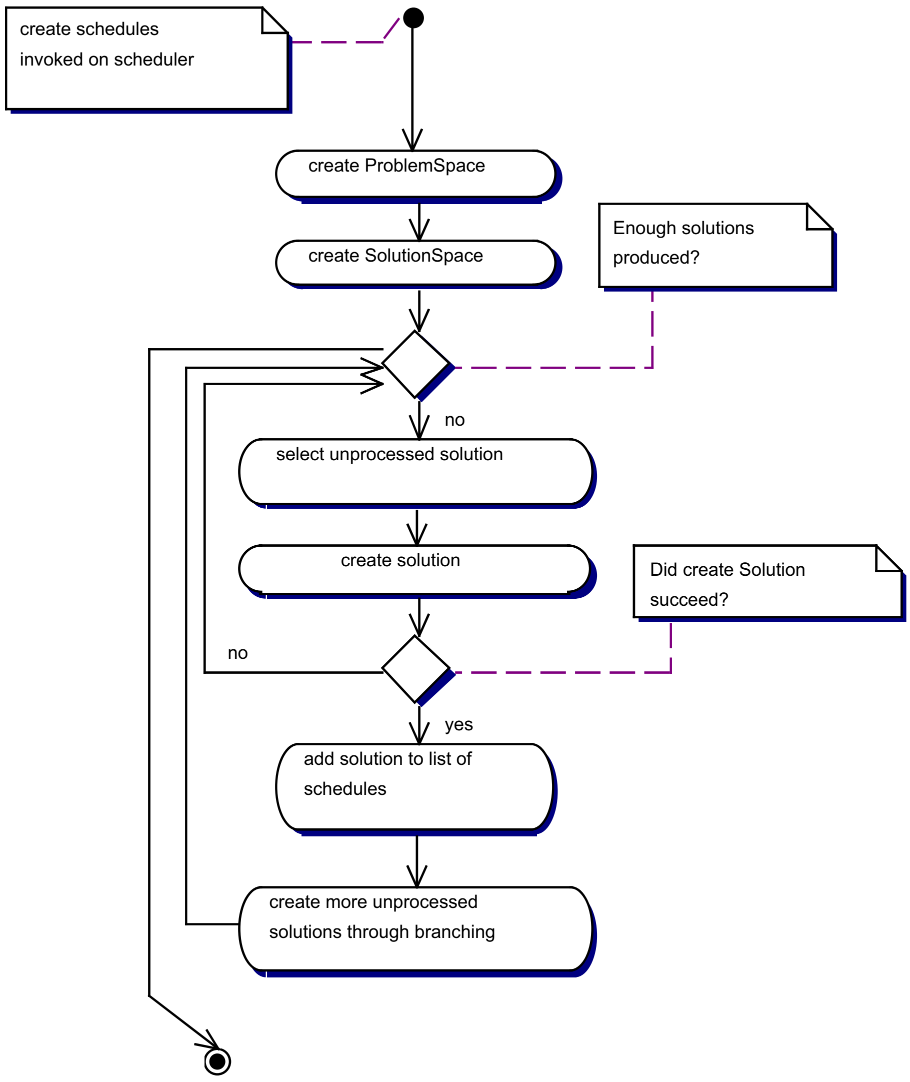

create schedules invoked on scheduler
create ProblemSpace
create SolutionSpace
Enough solutions produced?
no
select unprocessed solution
create solution
Did create Solution succeed?
no
yes
add solution to list of schedules
create more unprocessed solutions through branching

Figure 3.27: Activity diagram of the scheduler's strategy of improving a schedule.

over time and to decide for himself when and which schedule to take. The user may request the collection of created schedules at any time to see if there are new schedules available.

Efficiency An important factor for the runtime of the algorithms is the size of the OptionsPool. Let s be the size of the OptionsPool. The OptionsPool contains time slots for each non-dynamic resource that occurs in the job. For each of these resources, several instances may exist. An instance is a resource that matches the resource that was described in the job, and that exists somewhere on the Grid. For each instance, time slots are requested. The number of possible time slots per instance is limited by the time slot the job is supposed to run within:

Suppose the job's time slot covers t time-units.
The maximum number of different time slots at which that resource is available is $t/2$. For each of these time slots, which must be at least one time unit in length, there must be another time slot at which the resource is not available. Otherwise, two time slots could be merged. It can not be predicted how many instances of a specific resource will exist on the Grid. Assuming that for each resource r_i there are n_i instances, each having $t/2$ different time slots, the OptionsPool may contain up to $\sum n_i * t/2$ time slots. In order to be able to continue the estimation of runtime the following is assumed:

There is only one instance per resource, having $t/2$ time slots. Let r be the number of non-dynamic resources defined in the job. Then, s would be $r * t/2$.

(a) **Creating a schedule** This Algorithm (See Figure: 3.24) performs a search over all possible combinations of time slots of different resources that can be found in the OptionsPool. That means that in the worst-case (if not one combination is valid) all those combinations are tried out before the algorithm fails. Assuming that the OptionsPool contains $t/2$ time slots for r different resources, there will be no less than $(t/2)^r$ possible combinations to try out.

(b) **Finding a good schedule** Every time a valid solution has been produced, one of its selected, non-dynamic resources is excluded, so that no child-solution will be the same as the actual one. However, since the LowerBoundSolver always produces combinations of resources in the same way, all the combinations that were already discarded in the last run of the LowerBoundSolver will be tried again before new combinations of time slots are tested in a new run. So every new computation of a schedule will take longer than the last one.

    Assume there are no limits on how many iterations the scheduler may perform to find a good schedule.
    Let c be the number of possible combinations that the LowerBoundSolver will produce. In the worst case, each of these combinations is a valid schedule. In that case the scheduler would do c iterations. Let x be the current iteration. To produce the combination of time slots for iteration $x + 1$ the LowerBoundSolver would have to reproduce the combinations of iteration 0 to x. Doing c iterations, the computation of

$$\sum x = c * (c/2 + 1) = O(x^2)$$

    are accumulated. Assuming that c is $(t/2)^r$ the estimated worst-case runtime is

$$< O((t/2)^{2*r}).$$

Summary For the prototype implementation, a very basic scheduler was implemented that, if given enough time, can produce the optimal combination of given instances. It handles some of the DoItems a job may contain, but not all, so that only very simple jobs can be scheduled.

The behavior of the OptionsPool results in the generation of a lot of network traffic before scheduling actually begins. This strategy may be inefficient because most of the options generated will most likely never be touched since the first valid solution generated will be returned by the LowerBoundSolver. It might be wiser to keep resource instances first and request time slots on demand only.

Also, many options may be very similar with regard to the resource and the time slot they represent. The overall complexity of finding one schedule can be reduced by dividing options into classes and keeping only a certain number of options per class. These classes could be induced by a yet to find equivalence-relation.

Furthermore, the strategy of the SolutionSpace in finding a better schedule is merely a local-search which may easily fall victim to local optima. Its general convergence speed can be considered poor, too.
Ways around this might be resorting to a more randomized approach like genetic algorithms.

Originally it was intended to use a branch&bound approach. The methods for branching and the creation of a lower-bound are already there. The computation of an upper bound however proved to be problematic. Since the objective-function can take any form it is difficult to compute the maximum-value this function could possibly produce.

As stated before, it is differentiated between dynamic and non-dynamic resources. It was also decided to generally treat data-transfer and network-resources as dynamic.
This is already a big limitation to the flexibility of the scheduler. It might well make sense to ask questions like: “What machines with the following characteristics can this file be transferred to

until this time?” or “Give me all machines with the following characteristics that can be connected through a 100megabit connection with a latency below 100 milliseconds at a given time slot.” In such a case network and data-transfer resources would not be dynamic, but the choice of Bob Resources would depend on them. The interface is currently not powerful enough to handle such questions, neither are the modules responsible for data and network-management capable of answering to such requests. Figure: 3.28 shows an overview of the classes involved.

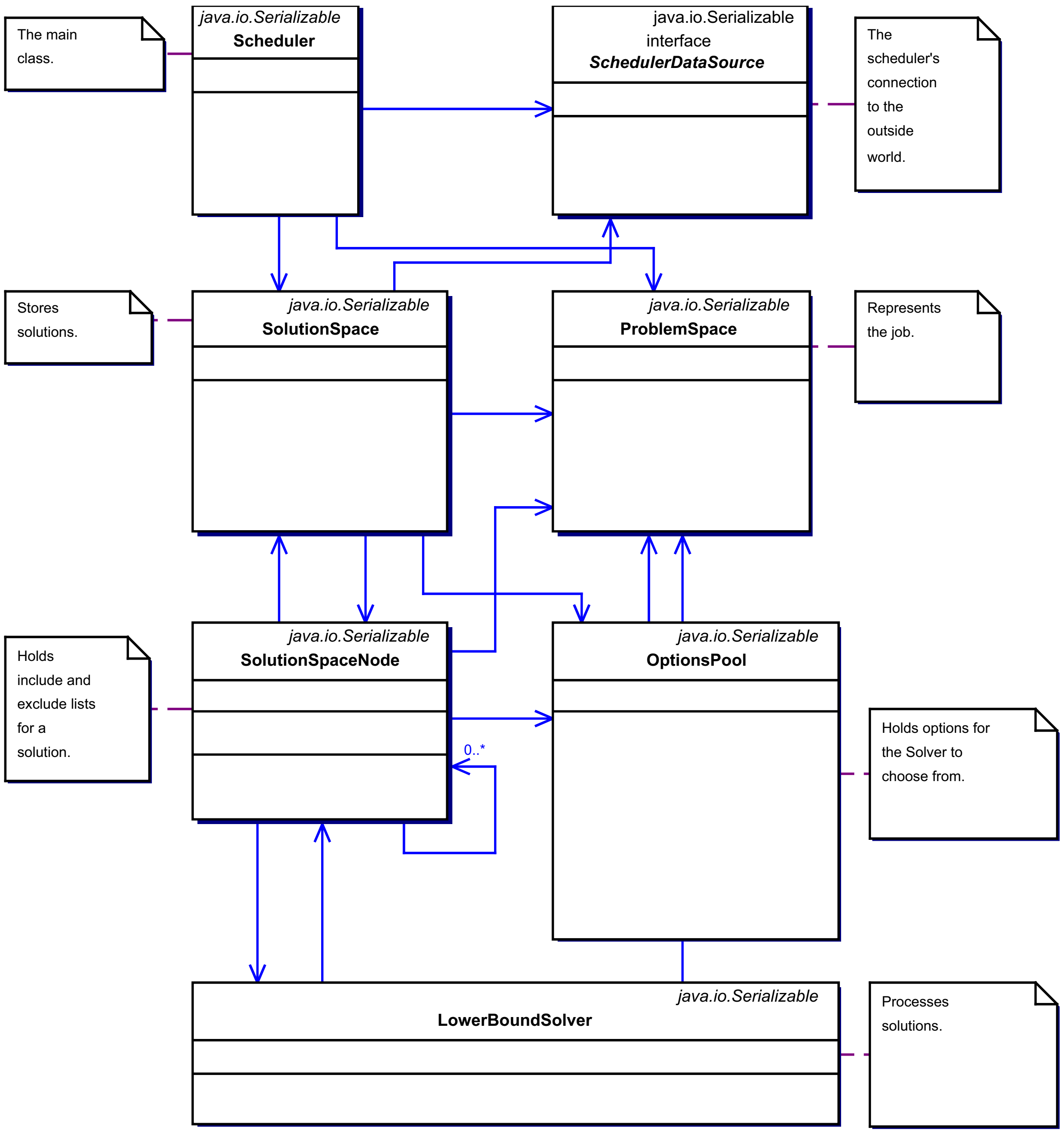


Figure 3.28: Overview of the scheduler module’s classes.

3.4.1.2 NuSched

A Flexible Framework for Super-Schedulers

Idea From the experiences with the SimpleSched the following weaknesses were identified:

- **Strict, hard coded order for selecting resources.** SimpleSched imposes a special sequence in which the resources of a job had to be selected.This order must be known to the implementing programmer and be hard coded into the scheduler. So SimpleSched's code must be altered on the introduction of every new resource and DoItem. The order in which resources must be queried however is neither necessarily obvious, nor must it be unique. It is thus desirable that a scheduler makes as few assumptions as possible regarding the order in which resources should be selected.
- **Resources and dependencies are implemented at object level.** Some properties of a resources description are dependent on the resources that have been selected before. Other properties in turn influence the description of resources which have not yet been selected. For SimpleSched, this problem was solved in a rather inconvenient way:

  For each resource, a separate class was implemented and some of its get-methods where overwritten so that they accessed the properties of other problems which they depended on.

  Example:
  A network connection's get-methods for the names of the start- and end-nodes to connect to would access the corresponding computer resources 'uri' property and return their corresponding hostnames.

  This approach proved to be error prone and inflexible. The kinds of properties that would require such redirection had to be known in advance and had to be implemented manually. It may therefore be wise to have a more general model of resources which allows flexible manipulation and analysis of its properties and their dependencies.
- **No validity checking.** When SimpleSched queries the Grid for an offer on a resource, no validation is performed whether the request makes sense at all, or if the reply received even satisfies the request. This could lead to very strange behaviour which could greatly endanger the robustness of the scheduling algorithm. It was evident that a convenient way of ensuring the validity of requests and replies.
- **Monolithic design.** As SimpleSched's design was so straight forward, many features where crammed into relatively few classes which soon exceeded their original scope of competence. SimpleSched became difficult to debug and to extend as more and more features where added during the course of its implementation. A new approach should feature a clean design that is optimized for flexibility and expandability.

Design To overcome these weaknesses a more general approach was considered. First of all, a data structure that would conveniently reflect inter-resource dependencies had to be designed. Furthermore it had to be flexible enough to handle new features and new kinds of resources and it should be independent on the scheduling algorithm's implementation as well as on concrete Grid access mechanisms.

A flexible data-structure for schedules: The Multi-Layered Graph It was discovered that a schedule can to some extend be regarded as a graph. Resources form the **Nodes**[2], while their inter-dependency can be interpreted as **Edges**. In the following, it is enumerated on the concepts and features of the data-structure that was designed for NuSched.

- **Hierarchical precedence of properties: The Multi Layered Graph** Initially, a Schedule contains just the blueprints for the associated resources. Normally, the user would not define all required properties of all resources.

  Usually the timeslot that is common to all resources that are to be employed together are defined by the parent target of those resources. So there should be some way to define those values independently on the resources' blueprints. If however the user entered a timeslot for some of the job's resources those values would have to override the more general values from the parent target.

[2]In the rest of this section the terms "Node", "Edge", "Field" and "FieldSet" refer to entities of NuSched rather than to entities of the NetworkManager that also employs such terms.

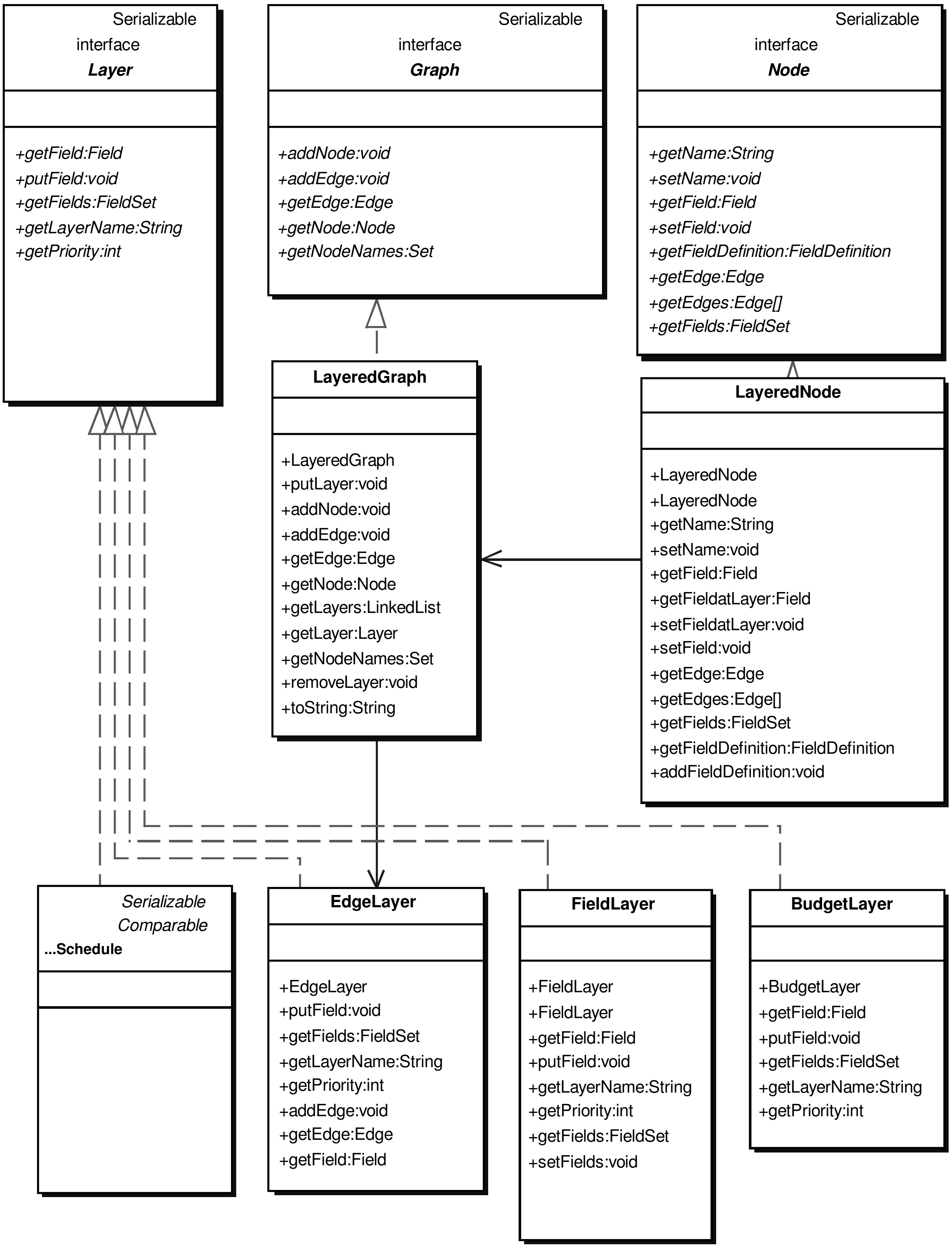


Figure 3.29: Classes of NuSched's 'Graph' package.

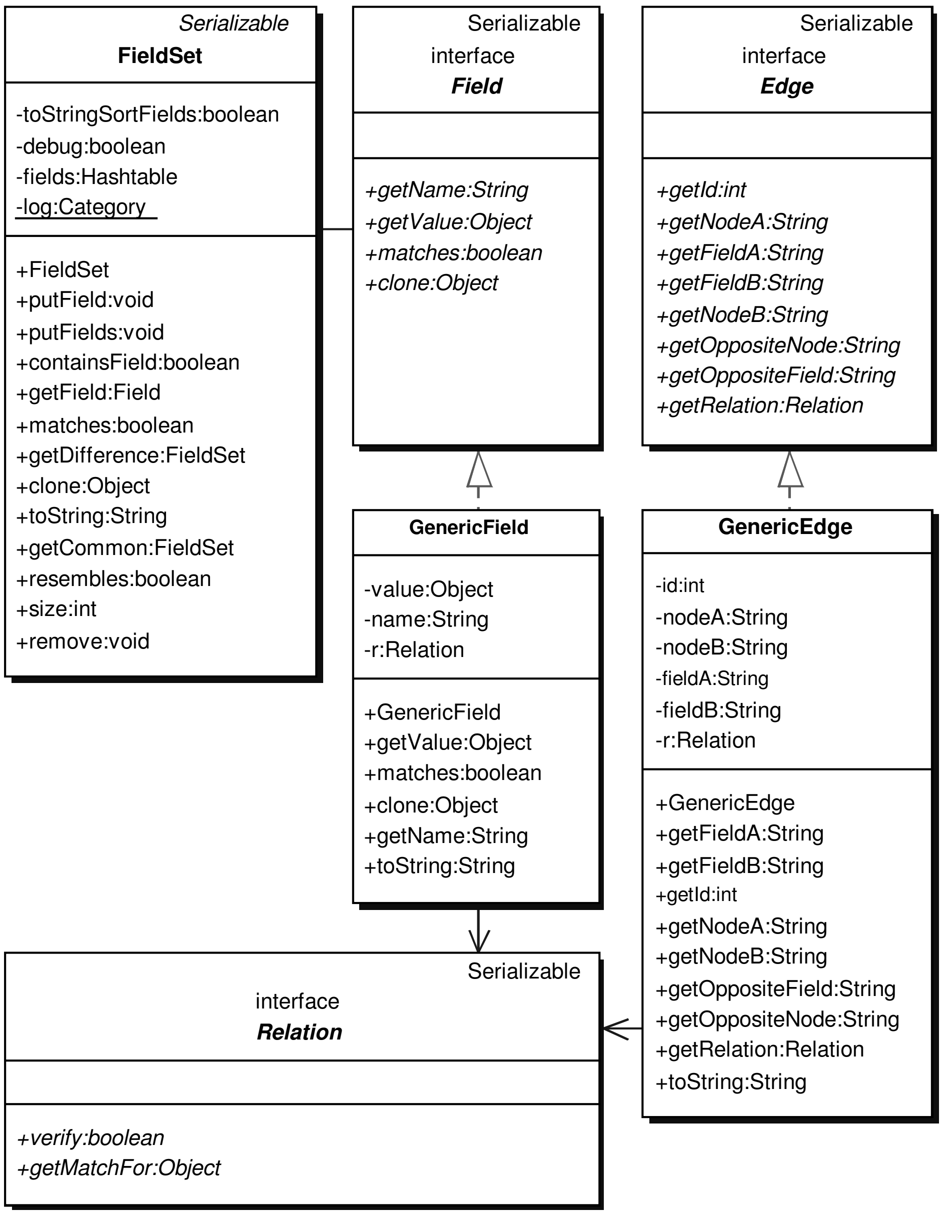


Figure 3.30: More classes of NuSched's 'Graph' package.

To provide such functionality the concept of layers is introduced. Each **Layer** contains some of the properties of some of the resources. In the Graph, multiple Layers are used. The lowest Layer contains default values for prices and timeslots of each node. It is called the default Layer. Above that Layer the values of the resource descriptions derived from the input job are stored. This is called the job Layer.

If the Graph is called to return a property it traverses its Layers from top to bottom until it finds a Layer that contains the desired property. When the Graph is called to retrieve all of a resources' properties, it searches them bottom up, so that properties from higher Layers replace those of lower Layers.

- **Abstract definition of resources and their properties: Fields, FieldSets and FieldLayer** A resource is understood as a collection of Fields which represent the resource's properties. Such a collection is called a **FieldSet**, a property is called a **Field**. An incomplete FieldSet can serve as a kind of blueprint that describes only those properties the desired resource needs to have. Comparing two FieldSets Field by Field allows a verification whether a resource found on the Grid matches a blueprint. Each Field has a name, so that corresponding Fields can be found in the FieldSets that need to be compared. The value of a Field can be just about anything, so each Field comes with a Format.

In the Graph, each Node represents a resource, and each resource is represented by its Fields.In the case of a FieldLayer Fields are stored in one FieldSet per Node. It provides methods to obtain all of a nodes Fields as a FieldSet or to fetch a single Field from a single Node by its field-name.

- **Comparing properties of resources: Relations** To allow operations like comparing Fields relations must be introduced. A Relation takes two Fields and returns whether or not those two Fields correlate or not. Additionally a method that takes one Field as an input and returns another Field so that this new Field is in relation to the old one was created. For each Format there is such a Relation. Each Relation comes with a relation-type. A relation-type can be common relation-symbols like equality, less than, more than... but they can be also very special.

  Imagine a relation for an URL-Format. Two URLs are to be related when their hostnames are the same. So a relation-type named "same_host" is created.

- **Realizing dependencies between properties through edges: The Edge Layer** Some properties of a resource depend on the properties of other resources.

  Example:

  If some resource X was to be employed at the same time as some other resource Y, then it its obvious that in the schedule, the start- and endtime for resource X must be the same as the start- and endtime of resource Y.

  An Edge consists of two Fields and a Relation that describes how the two Fields depend on each other. It is possible to reuse the Relations that were created for comparing Fields. Edges are stored in a special Layer, the Edge Layer, which is usually situated above the job Layer.

  When a Field is accessed and there is an Edge on that Edge Layer which connects to that Field, that access is redirected accordingly until a value is discovered. Then, the relation associated with that Edge is applied to return a corresponding value. Redirection can take place multiple times since it is possible that the destination Field of an Edge is connected to even more Edges. After a finite number of redirections the following condition will occur: A Field is reached that connects only to Edges that have already been traversed. This can either be because of this Field being a dead end or the Field is part of a loop. Either way, Edge redirection is no longer possible and the search for the value of that Field is continued on the next lower Layer.

- **Scheduling: The Scheduling Layer** Any scheduler will consult the Grid for resources. Each reply is converted into a FieldSet and is stored in the scheduling Layer, which is conceptually a Field Layer. The scheduling Layer is placed on top of all other Layers, because there is no way these values can be altered but by re-requesting a resource. The requests will be made according to the Fields procured from the Layered Graph, which automatically computes all of a resources properties with the help of its Layers. All dependencies are applied as soon as values are retrieved from the Graph and any change the scheduling algorithm performs on its schedule immediately influences the properties of resources which are yet to be selected.

- **Computing price constraints: The Budget Layer** Selecting resources must not exceed the budget entered for the job. Therefore, requests for resource offers provide a property that define the maximum amount that can be spent on the reservation of that resource. The remaining budget however depends on the prices of those resources that have already been selected and the original budget itself, obviously. To compute the maximum budget available to one remaining resource, the concept of the budget Layer is introduced. It is situated below the Edge Layer and above the job Layer. This Layer, when queried for the price property of a node will take the original budget and subtract the prices' property values of all resources selected so far and return the result.

The reader will possibly realize that this concept is indeed powerful and flexible. There is no limitation on the number of Layers which can be added to the Graph, and virtually any information can be stored and modified regardless of format and purpose. The functionality of each Layer is not limited to just storing data, its behaviour can be complex like that of the Edge Layer or the budget Layer.

Having found a data-structure that seemed to meet the requirements the other components necessary for NuSched can be specified.

Converting GridSched resources into FieldSets and back: The Analyzer To convert a job into a multilayered-Graph and back, and to translate between GridSched resources and FieldSets, a package named analyzer was designed.

The GraphFactory contains Methods for creating a Graph from a Job and vice versa, which includes traversing and analyzing the target and resource hierarchy of a GridSched job and setting up the Graphs Nodes accordingly. Additionally it inspects a job's do Items and creates the appropriate Edges.

Resources are converted using ResourceConverters by processing a corresponding set of FieldDefinitions. Each FieldDefinition holds information on how to get and set a named value of a given resource, what it's format does look like and what the resulting Fields relation should be. Since GridSched Resources mostly store primitive data-types instead of abstract containers like NuSched's Field objects, special rules were established to decide when a resources property was to be regarded as 'undefined'.

Example:
When a job's computer resource's number_of_cpus property is zero, that value is regarded as undefined instead of forcing the scheduler to look for a computer that does not have any CPU.

Edge creation is done by assigning Edges recursively for each DoItem. For that purpose a class named StaticDoItemRules holds instructions on which Edges to create for a given DoItem. Converting a Graph back into a Job is currently done by replacing the corresponding resources' properties with the properties found in the Graph.

With help of the Graph package and the analyzer package it became easy to inspect and verify jobs and their resources. An implementation of a job verifier can be found in
`org.gridsched.scheduler.JobVerifier` .

Accessing the Grid: The Grid Agents Each resource that takes part in a scheduling process is assigned one GridAgent which serves two purposes.

(a) When queried, it returns a resource found on the Grid that matches the requirements formed by the current state of the Graph.

(b) The Agent itself decides which resource out of a pool of possible options it should return. It uses caches to keep communication with the Grid to a minimum.

This leaves a great degree of freedom to the implementing programmer on how to design the agents. In the following it is elaborated on how implemented Grid Agents were implemented and what their respective features are.

- **Cross schedule validity** While one resource is assigned to exactly one agent, one agent may be assigned to multiple resources at a time. If a job requires multiple instance of one resource this prevents repeated requests to Grid. It is even possible to assign resources of different jobs to one agent. If multiple, similar scheduling processes are running in parallel this can cut communication cost.
- **Fully event driven: Introspection** To allow effective handling of many requests in parallel, the Grid Agents and their components are entirely event driven. This was accomplished by use of the Introspection package which was designed for GridSched. See
  `org.gridsched.util.introspection` for details. Each component that is part of a GridAgent roughly works the following way:

  Events are synchronized by use of an event queue. For every incoming event that is dequeued-queued, the appropriate modifications are performed on the data-structures of that component.

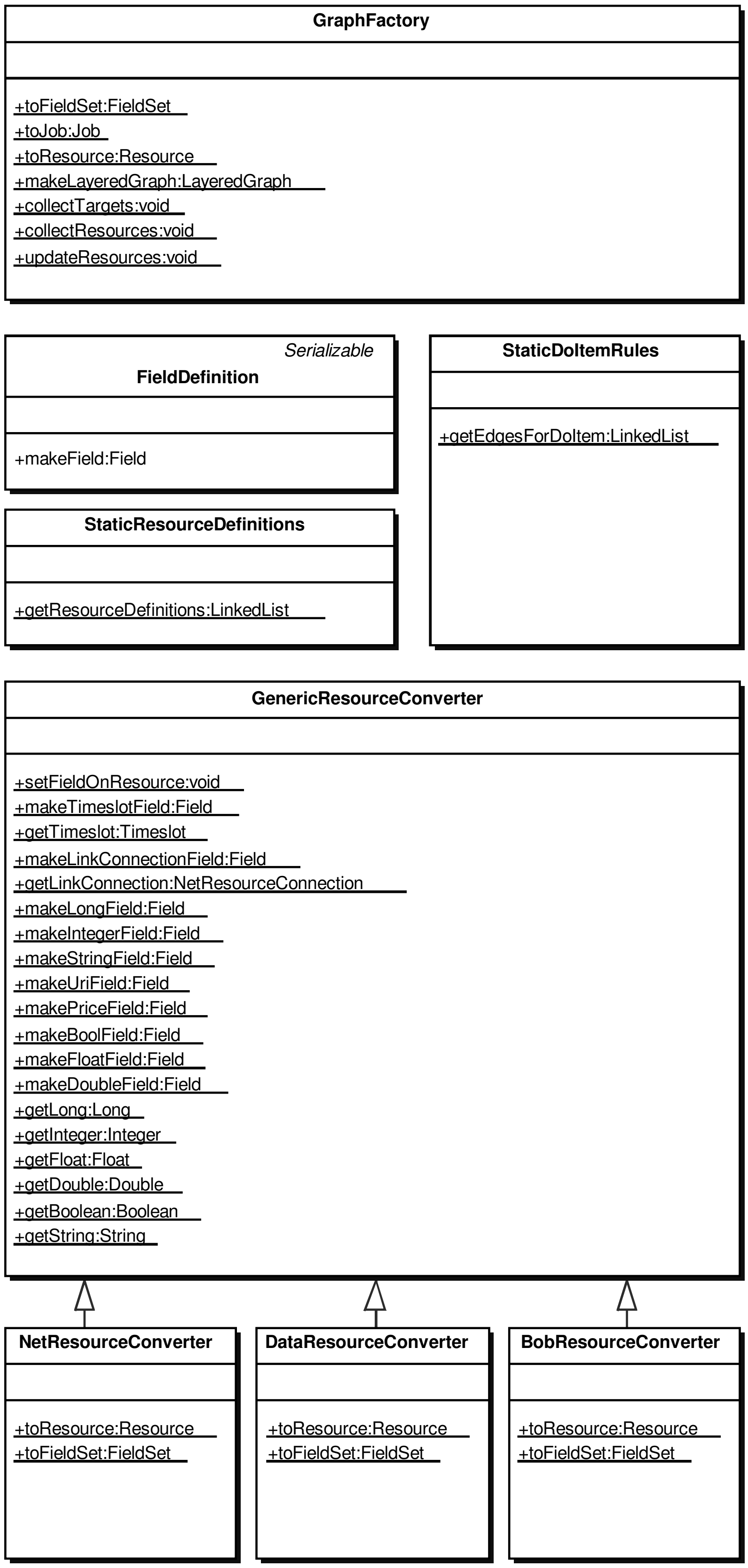


Figure 3.31: Classes of NuSched's 'Analyzer' package.

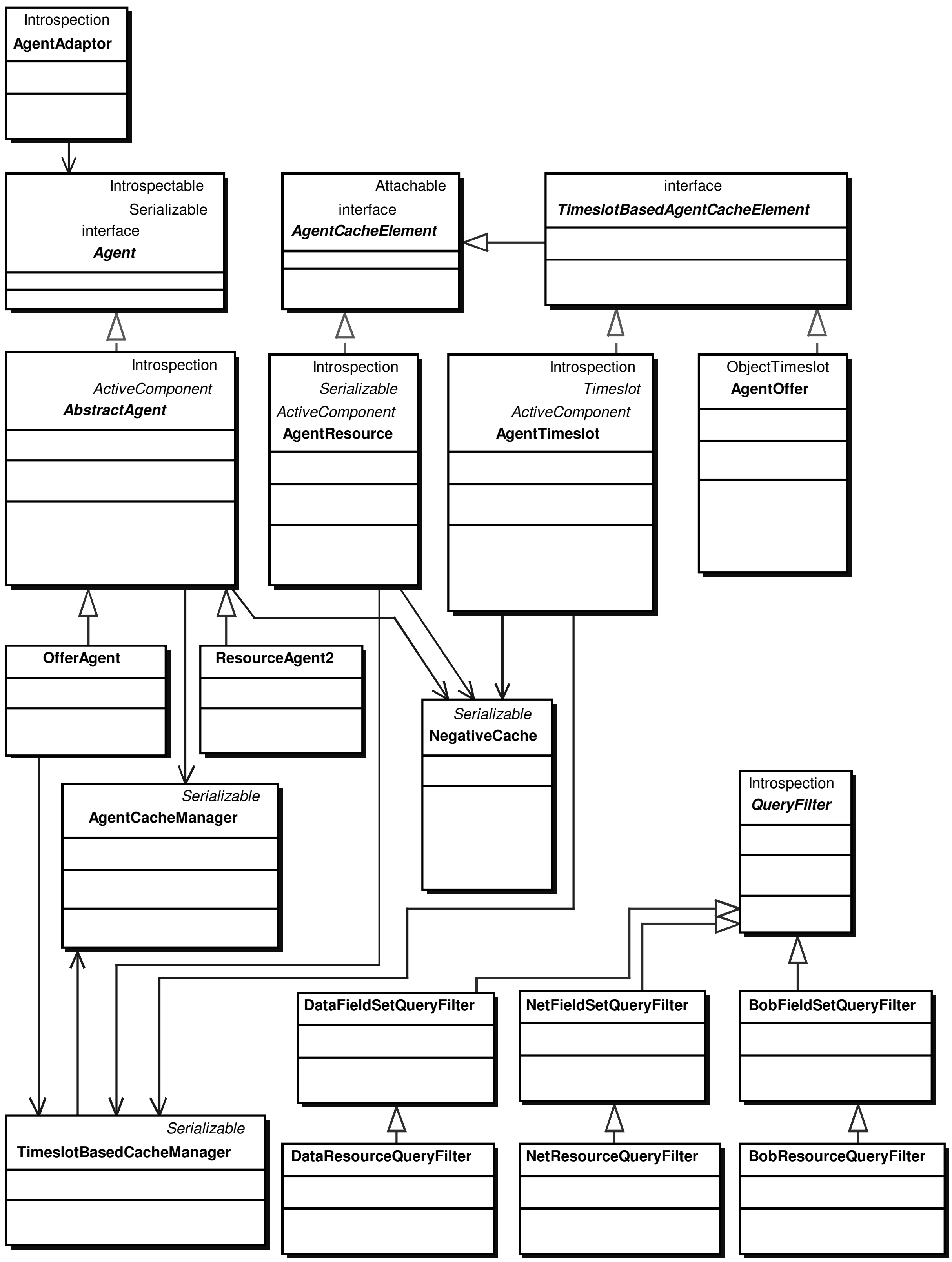


Figure 3.32: Classes of NuSched's 'GridAgent' package.

Afterwards a central method is invoked that inspects the new state of the component and triggers new events accordingly. (See Figure: 3.33)

- **Time constrained caching strategies** In Grid Scheduling, special care must be taken on how long an information fetched from the Grid will be valid. It is most likely that multiple scheduling processes are considering the same resources for their schedules. A cached resource's information can become invalid, because in the meantime reservations have taken place so that the resource is no longer available. For that reason, all entries in the GridAgents' caches do only have a limited time of validity and are purged from the cache once that span of time expires.
- **Offer returning strategy** Each entity of the GridAgent that is faced with multiple options to handle a request can employ a depth first, or a breadth first strategy. In the first case, it would evaluate only one option at a time and proceed once the results about the current option are available. In depth first mode, options that have been considered the least are tried before others. In breadth first mode, all options are evaluated in parallel, since those options are usually represented by other GridAgent entities it depends on which of those entities takes the least time to evaluate the option and to return the desired reply.

An AgentFactory manages a pool of Agents and returns one instance of an Agent according to the kind of resource that the agent is to be attached to.

Currently, there are two different kinds of agents, ResourceAgents and OfferAgents.
ResourceAgents are used for resources that allow discovery by a remote directory service (See Section: 3.5). They employ a three level caching scheme: (Resources, Timeslots, Offers). So there are entries that store information on one existing Grid Resource each. Every such entry may refer to multiple entries which store information on the different timeslots when that specific resource is available. Those entries in turn maintain information on the different offers that where procured for those individual timeslot.
Each such entry is an active component of the corresponding ResourceAgent which handles and forwards requests independently.
OfferAgents are responsible for such kinds of resources where discovery is performed otherwise and which cannot respond to timeslot requests. For those kinds of resources, only offers can be cached.

Finding a good schedule: The Algorithm

- **Finding one schedule: The Solver** To produce a schedule, the solver maintains four stacks: (toDo, pending, selected, failed). Initially, all resources that need to be scheduled are placed on the toDo stack. In contrast to SimpleSched, NuSched makes no assumptions on the order in which resources must be selected. Therefore, if not specified otherwise(i.e. by the optimizer) the selection order is created in the following way:

  For each resource, the corresponding agent is called to return a value on how many options it can possibly produce for the resource in question. Resources on the 'toDo' stack are then sorted in ascending order.

  In every iteration, one resource is taken from the 'toDo' stack and the corresponding agent is contacted to provide an offer for the employment of that resource. In the meantime, the resource is placed on the 'pending' stack. Currently, there may only be one resource on that stack, but theoretically the solver may request multiple resources in parallel if those resources are independent. Once the agent in charge returns, two conditions may occur.

  (a) **The query could not be processed successfully.** This happens if no resource could be found that matches the conditions formed by the current schedule. The resource is moved from the 'pending' stack to the 'failed' stack.

    – If there are items left on the 'toDo' stack, the next resource is taken from that stack and the corresponding agent is contacted to return an option.
    Reason: Selection of a resource failed. The reason may be that there is simply not enough information available to make a meaningful request for that resource. It is

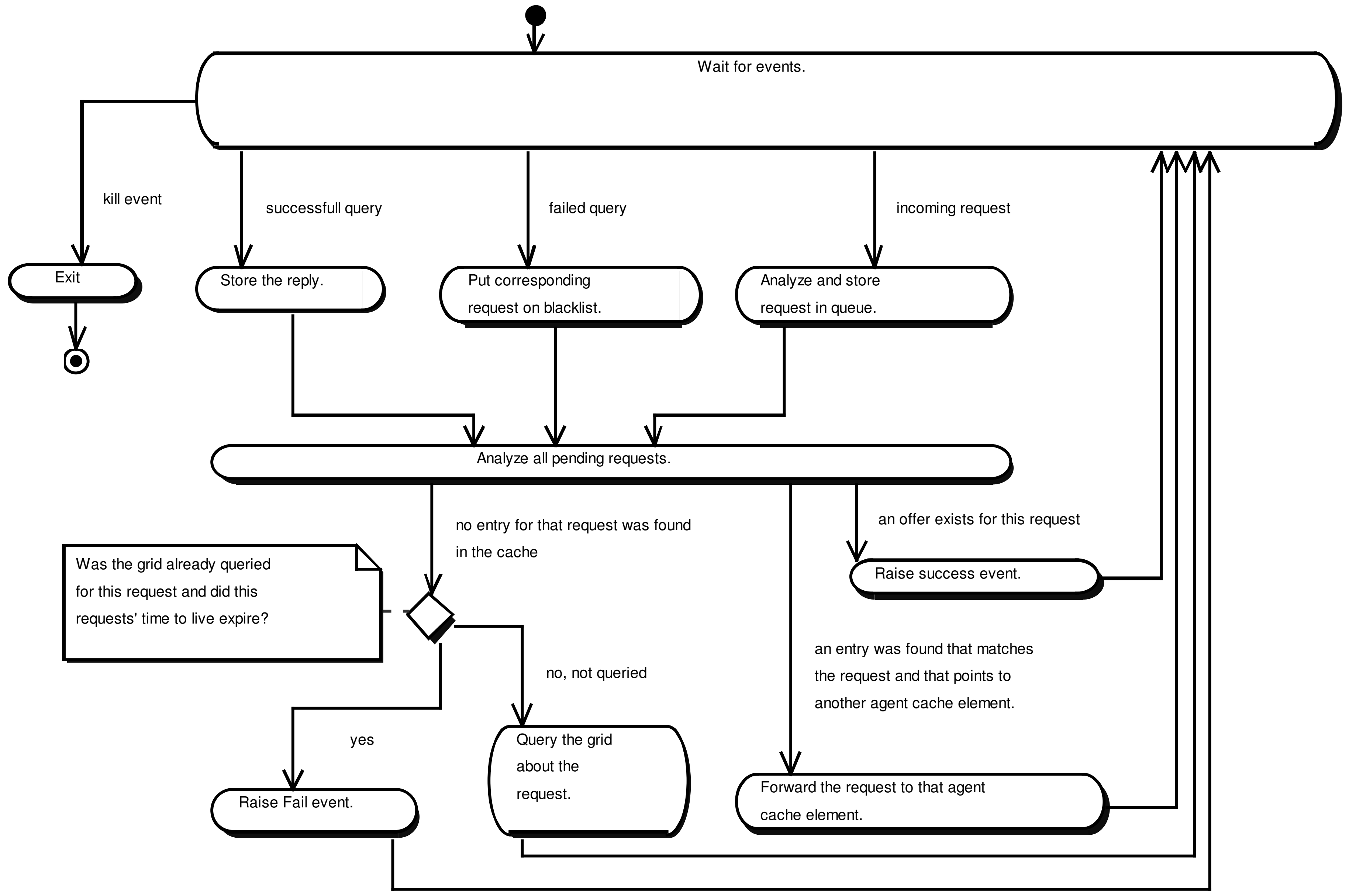

Wait for events.
kill event
successfull query
failed query
incoming request
Exit
Store the reply.
Put corresponding request on blacklist.
Analyze and store request in queue.
Analyze all pending requests.
no entry for that request was found in the cache
an offer exists for this request
Raise success event.
Was the grid already queried for this request and did this requests' time to live expire?
no, not queried
an entry was found that matches the request and that points to another agent cache element.
yes
Query the grid about the request.
Raise Fail event.
Forward the request to that agent cache element.

Figure 3.33: The basic workings of every implemented GridAgent component.

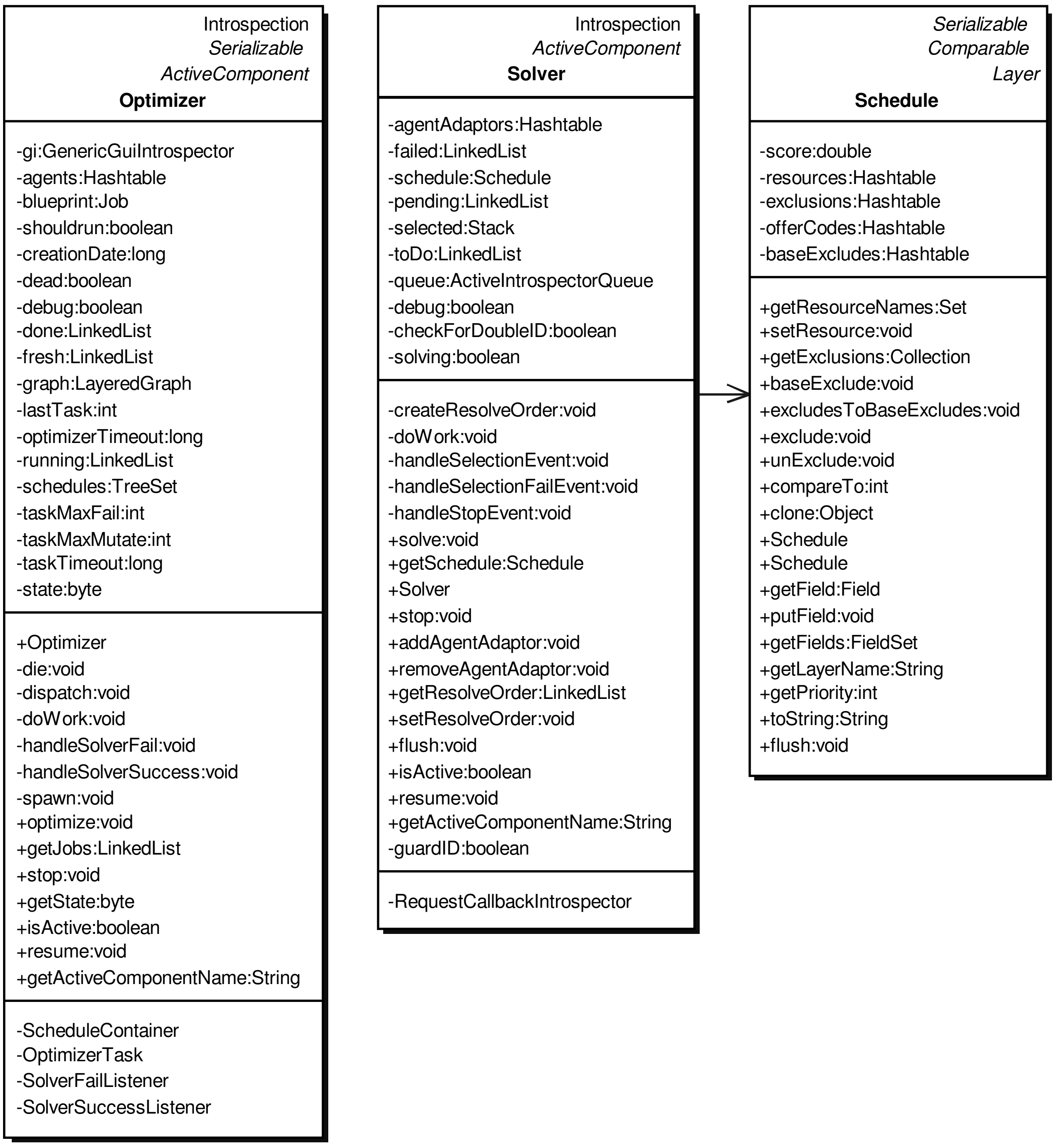


Figure 3.34: Classes of the 'Algorithm' package of NuSched.

tried to select some of the other un-processed resources first before it is tried to select that resource again.

  - If there are no resources left on the 'toDo' stack, but the 'selected' stack is not empty, the resource on top of that stack is removed, its current option is excluded and the corresponding agent is asked to return another option for that resource. The resource is moved to the top of the 'pending' stack.
    Reason: Requesting a resource failed because the constellation of the resources already selected made it impossible to select any other resource. The algorithm must fall back and create a different constellation so that all resources can be selected successfully.
  - If all resources are on the 'failed' stack, the solver has failed to produce a schedule.
    Reason: The algorithm has tried all combinations of resources possible under the given ordering of requests, but there was at least one resource that could not be selected independent on the constellation of the other resources.

(b) **The query could be processed successfully.** The resource is moved from the 'pending' stack onto the 'selected' stack. All resources from the 'failed' stack are moved back onto the 'toDo' stack. If there are no resources on the 'failed' stack, the schedule was completed successfully.
Reason: Requests that did fail may now succeed because the constellation of selected resources is now a different one. If all resources are selected there is nothing more to do.

- **Optimizing schedules: The Optimizer** The purpose of the optimizer is to create the best schedule possible. The creation of a valid schedule is called a task. A successful task may be modified and re-computed a given number of times in order to produce a better value, which is called 'mutation'. Even if a task fails, it may be modified a given number of times and be re-computed, which is called 'revival'.

  The Optimizer maintains a set of successful scheduling tasks(initially empty), as well as a list of un-processed scheduling tasks(initially one task). As long as no stop criterion applies the optimizer would perform the following:

  (a) Stop any running task that has expired its time quantum.
  (b) Dispatch any task on the list of un-processed tasks.

  Dispatching of a task consists of inserting the corresponding schedule-layer into the layered-Graph, so that selection of resources can take effect. Additionally the participating Agents are attached to the solver so that they can accept requests from it. Finally the solver itself is started.

  Upon the successful termination of a task, that task is moved to the set of successful tasks. Afterwards it is attempted to find a task from that set whose schedule produces the best objective value among all tasks and which has not been mutated more than the maximum number of mutations allowed. The resources selected for that tasks are excluded, so that re-running that task will not produce an identical schedule. The newly mutated task is en-queued for dispatching.

  If no such task can be found, the optimizing process is declared to be over. If there is at least one task in the set of successful tasks, the process is declared successful.
  If a task should fail, be it because of timeout or because the corresponding solver could not find a solution, one of two possible actions are performed:
  If the task's maximum fail count has not yet been reached, that task is 'revived' by modifying its selection order. This revived task is en-queued for dispatching
  Otherwise that task is discarded and it is attempted to select another task from the set of successful tasks and to create a mutation of it.

Summary As expected, the introduction of a powerful data-structure greatly facilitated writing and expanding NuSched. Adding new resources is as simple as providing another set of FieldDefinitions for that new resources' properties. Adding different DoItems can easily be achieved by creating the appropriate Edge-creation-rules. The scheduling algorithm, the data-structure and the Grid-access-mechanism are well separated and can be exchanged almost seamlessly.

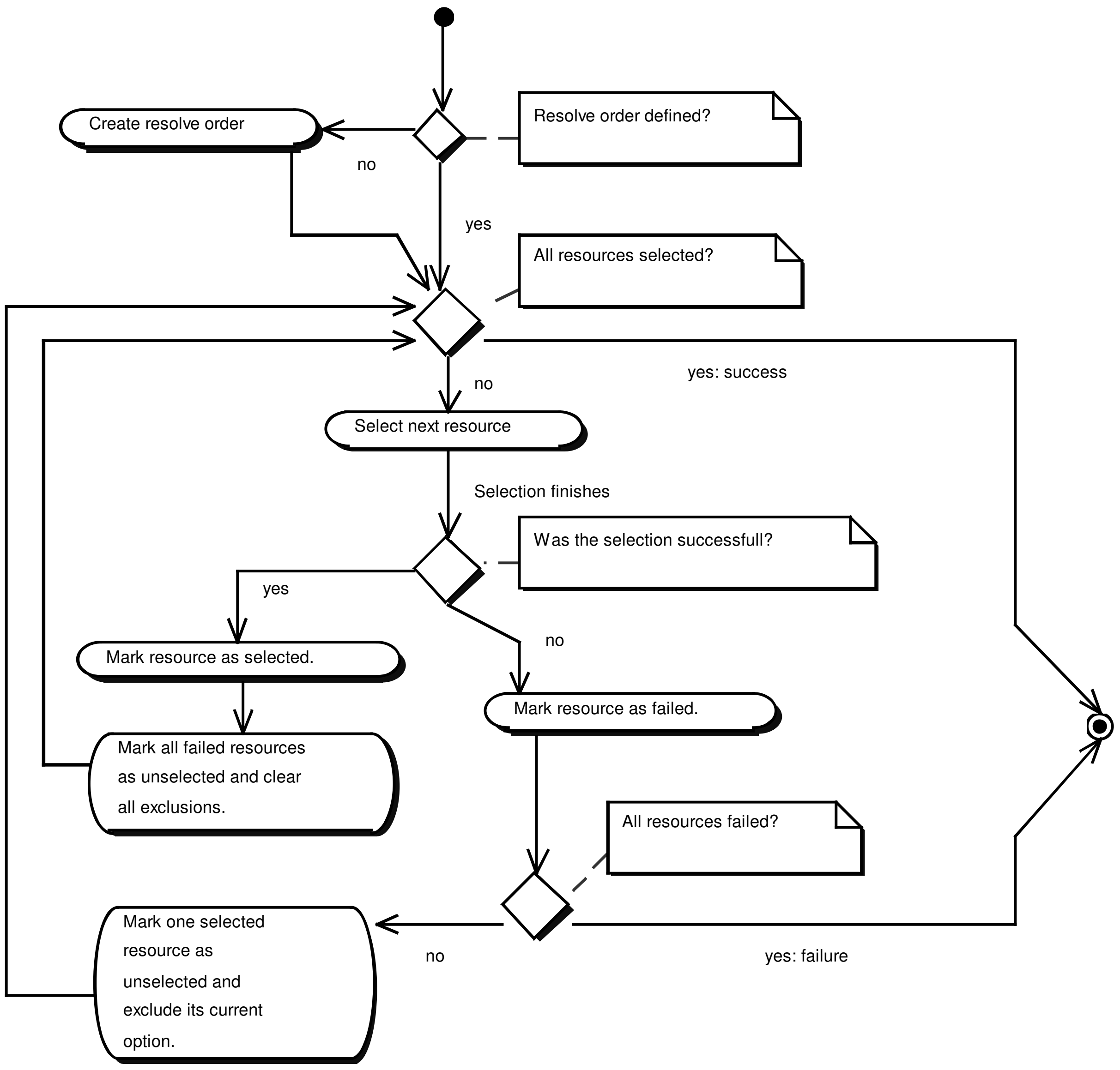

Create resolve order
Resolve order defined?
no
yes
All resources selected?
no
yes: success
Select next resource
Selection finishes
Was the selection successfull?
yes
no
Mark resource as selected.
Mark resource as failed.
Mark all failed resources as unselected and clear all exclusions.
All resources failed?
Mark one selected resource as unselected and exclude its current option.
no
yes: failure

Figure 3.35: Activity diagram of the solver's algorithm.

It is difficult to estimate the performance of NuSched in terms of O-notation, because most operations are constrained through timeout-values and the overall flow of control is less obvious as SimpleSched's.

One big issue remains in form of the current GridAgents. Since they were designed to be entirely event driven, independent active entities, there are a lot of threads competing for the CPU and a lot of synchronization and even more method invocations are performed even for the most simplest tasks. Deleting an entry from the caches not only consists of removing some references, but incurs a large overhead of stopping event queues, withdrawing event listeners and other clean up operations which became increasingly difficult do manage. The result is poor overall performance.

A better agent design would greatly improve performance and capabilities of NuSched, but at the moment of implementation insufficient knowledge on cache design and event driven programming was available overcome the weaknesses encountered.

Nevertheless NuSched is believed to have some potential. Among the ideas which where not realized due to lack of time are:

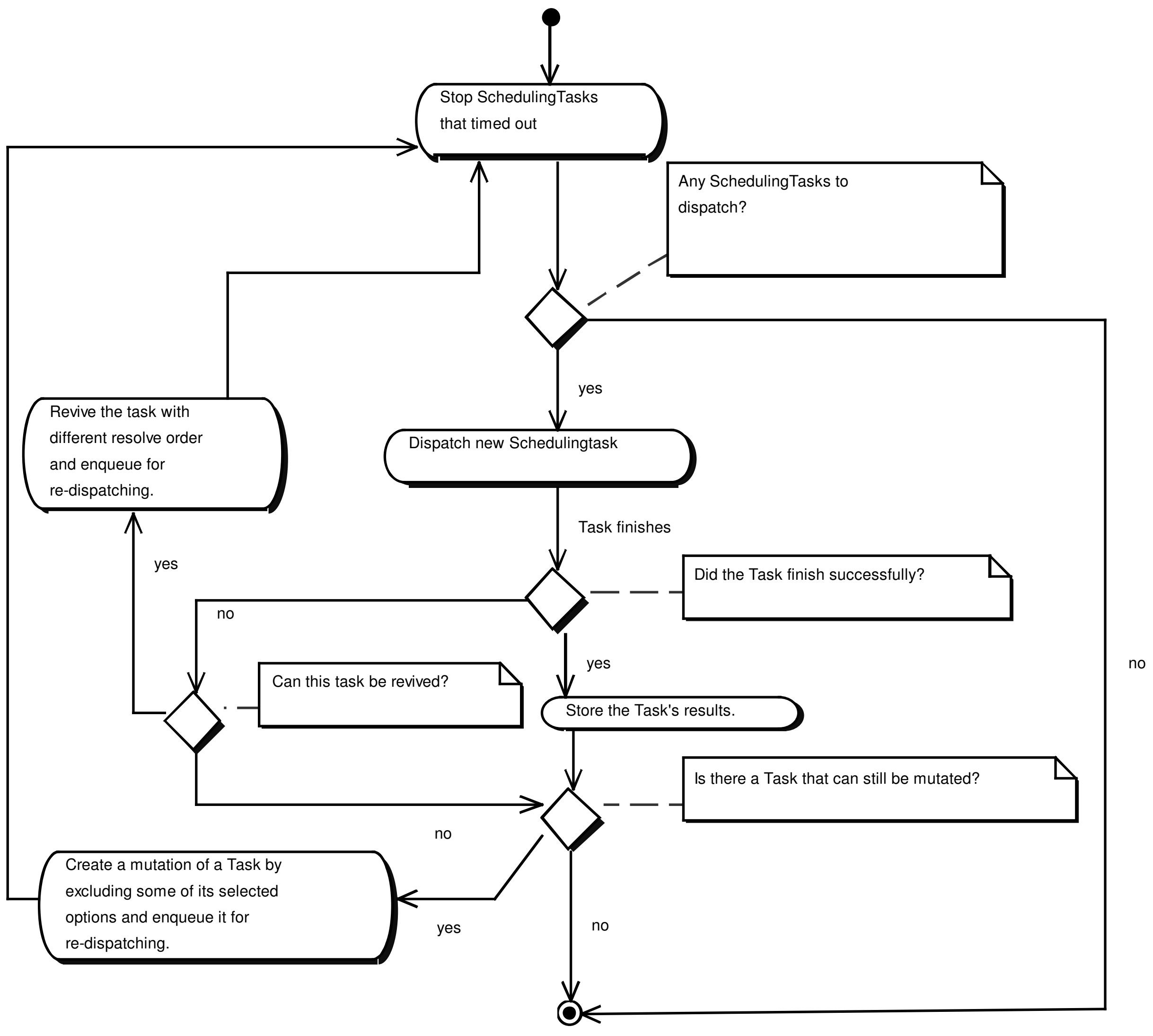

Stop SchedulingTasks that timed out
Any SchedulingTasks to dispatch?
yes
Revive the task with different resolve order and enqueue for re-dispatching.
Dispatch new Schedulingtask
Task finishes
yes
Did the Task finish successfully?
no
yes
no
Can this task be revived?
Store the Task's results.
Is there a Task that can still be mutated?
no
Create a mutation of a Task by excluding some of its selected options and enqueue it for re-dispatching.
yes
no

Figure 3.36: Activity diagram of the optimizer's algorithm.

- **Multi-Stage-Solving** Currently, NuSched's solving algorithm picks options without considering their compatibility with other resources. Or, more precisely, it may pick resources which are very unlikely to be available at the same time that other resources are available. A solution to this might be another solver which influences a different Layer in the layered Graph. This solver could, with help of a more sophisticated implementation of GridAgents, analyze at which timeslots there is a high probability to find options for all needed resources. By enforcing these timeslots, the efficiency of the original solver would improve.
- **Multi-Level-Optimization** The current optimizer performs very simple attempts of modification in order to influence the the quality of the schedules' objective value. There exist ideas to introduce another Layer where an optimizer could pre-set properties of some resources' blueprints which it thinks are inherent for a better schedule. Such optimizations could be achieved by employing means of statistical analysis on the set of schedules that where already produced. Doing so, common properties of 'good' schedules might be discovered, or it may be possible to extrapolate some properties' values and enforce them for further schedules.

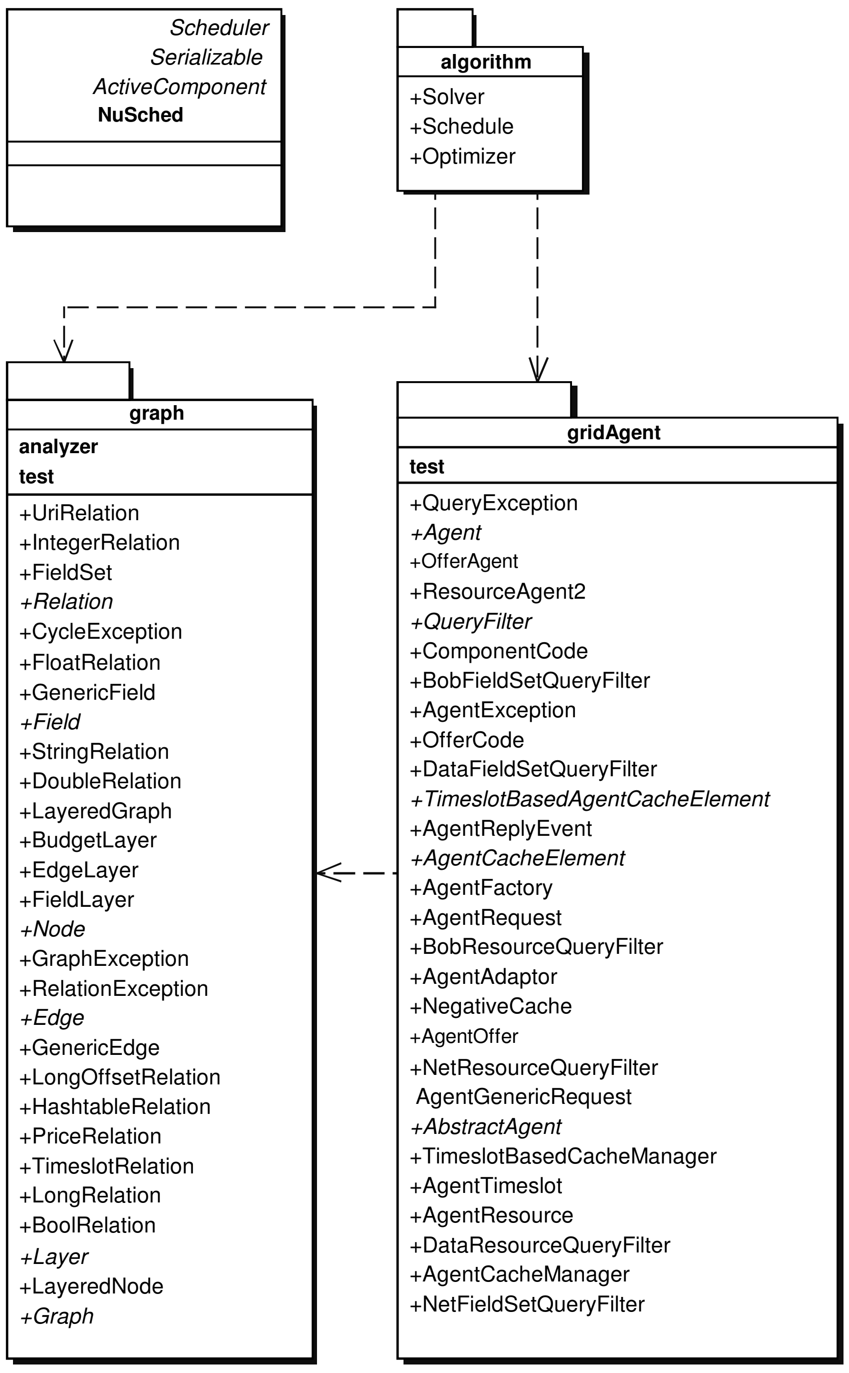


Figure 3.37: Packages of NuSched.

3.4.1.3 Conclusion

Performance Both schedulers can produce valid schedules but due to the nature of the optimization techniques employed, no good objective values can be expected if the number of options is large and the quality of resources varies. Too many iterations would be required to approximate a near optimal objective value. Another reason is the rather long communication overhead which results in long phases of waiting for replies during the course of scheduling. In some experiments, creating one schedule took more than 15 seconds even though there where no alternatives for the scheduler to choose from.

Internal tests with a predefined set of options showed that the schedulers can indeed produce dozens of schedules a second even if there are lots of alternatives to choose from. This huge gap shows that a much better way of inter-resource communication must be found in order to enable any kind of sophisticated scheduling.

Alternative Sets of Resource Definitions As presented in Section: 3.3.3.7, a DoItem may contain a term called 'or' that permits the consideration of alternative sets of resource combinations in a single schedule. Handling of this DoItem though is not currently implemented.

In SimpleSched, handling of DoItem - 'or' can be realized fairly easy since scheduling is performed by traversing the hierarchy of DoItems. If scheduling of one alternative of such a DoItem fails, the next option is attempted.

In NuSched, handling of such a DoItem can be realized by introducing different Layers for any alternative combination of resources. Deciding which of these alternative Layers will be effective in the current schedule can be done at the corresponding task's dispatch time, or it can be done during scheduling. In that case the decision is delayed until a resource that occurs in one of the alternative Layers is selected. Since that resource was selected, its corresponding Layer must be active and all other alternative Layers must be disabled.

Multi-Reservation of one resource Apart from performance, there is one more problem that might need further investigation. The scheduling algorithms must prevent that a resource is employed twice in the same schedule.

Consider a job that requires two computers which are to perform some kind of distributed computation. The scheduler must ensure that it does not select the same machine for the second computer as it chose for the first one. In general, this could be allowed if the machine in question has multiple processors, but in that case, the scheduler would still have to verify that BOTH processors are available at the same time. There are multiple ways to solve this issue.

(a) **Prevent any two resources in a schedule to have the same ID.** This is the strategy that was actually realized in SimpleSched and NuSched. If a resource is encountered that is about to be selected has the same ID as another resource that has already been selected, that second option is excluded and it is tried to find another resource with a different ID. This behaviour effectively prevents that a resource will be reserved beyond its capacity, but some promising options in scheduling are wasted.

(b) **Verify that each resource has enough capacity to handle the combined load.** If a resource is employed twice in a schedule, it must be verified that the resource can still handle the load. In other words, it must be assured that not just one, but all reservations for that resource will succeed. To do this the combined resources' needs must be computed and this request must be sent to the resource in order to find out if it is still available under those conditions. While this is technically possible, its realization would introduce more problems:

    Arithmetic operations on Fields must be implemented so that the combined properties of two resources can be computed. A similar technique was implemented for the TimeslotManager data-structure (See package: `org.gridsched.util.timeslot`), so this should be easy. Additionally it would have to be defined at scheduler-level WHICH of a resources Fields must be combined. For a computer-system resource the number of CPUs, the amount of memory- and mass-storage-consumption would have to be added. However that information must be available on every kind of resource. While this would be easy for the current version of GridSched(there are not too many different kinds of resources), this might not be a good idea for a more complex Grid environment with hundreds of kinds of resources and different policies concerning how multiple reservations add up.

(c) **Different scheduling approach, distributed super-scheduling.** Each local-scheduler must be sent a most recent copy of the current schedule instead of being sent the resource in question only. Based on that copy, each local-scheduler may inspect for himself whether or not it can grant a request based on what has already been selected in the schedule. Furthermore, if a local-scheduler is working on that schedule anyway, it might as well fill in as much of the resources still un-selected as possible. Doing so, it would then consider only those resources which it has control over. This way, competence of handling double reservation of resources is shifted towards the local-schedulers, because they possess the information available. The original super-scheduler would receive a partly completed, valid schedule as a reply and then request more resources by passing it to the next local-scheduler that might be able to fill in more of the un-selected resources.

This procedure could be realized by re-designing parts of NuSched, the communication Module, and of course the local-schedulers.

3.5 Resource Information Service

Grid Scheduling requires some kind of shared directory so that participating resources can be discovered. Schedulers may access that directory to find suitable resources for a given job. In the case of GridSched LDAP was chosen as a technology to register and discover participating resources.

Lightweight Directory Access Protocol (LDAP) is an open-standard protocol for accessing X.500 directory services. The protocol runs over Internet transport protocols, such as TCP. [15]

In order to store information on GridSched Resources in an LDAP directory, custom LDAP-schemes had to be created (See Section: 6.3.1). A frontend class was written to facilitate access to the directory of GridSched Resources. It employs SUN's JNDI-technology to interact with the remote directory service and offers Methods to publish and discover GridSched Resources.

Remote directory service interaction
To establish a connection to a remote service, an instance of the `JNDIProxy`, which supplies a set of operations on remote directories, must be defined. The constructor call has to be passed the designated network address, and the port number of the remote host. Before any operation can be performed, the method `connect()` must be invoked.

Publishing and discovering resources
The `JNDIProxy` methods
`publishNetResource()` and `publishComputeResource()` are to be used to add a resource to the connected directory server. The opposite are the calls `removeNetResource()` and removeComputeResource(), a remover or publisher must be proprietor of a resource, thus he must pass a resource object to any of these calls. To look up for resources, a search template in form of a resource instance must be instantiated. Each attribute occupied with a non-zero value becomes part of a string search pattern, this is done by the `BobGateway` class in the `org.gridsched.bob` package for e.g. .

The lookup calls
(`getMatchingNetResources()`, `getMatchingComputeResources()`) in the
`JNDIInterface` (See Figure 3.38) do only accept the resulting search string as parameter returning a `NamingEnumeration` instance of all matching resources, registered in the remote directory which are then transformed back into concrete resource objects.

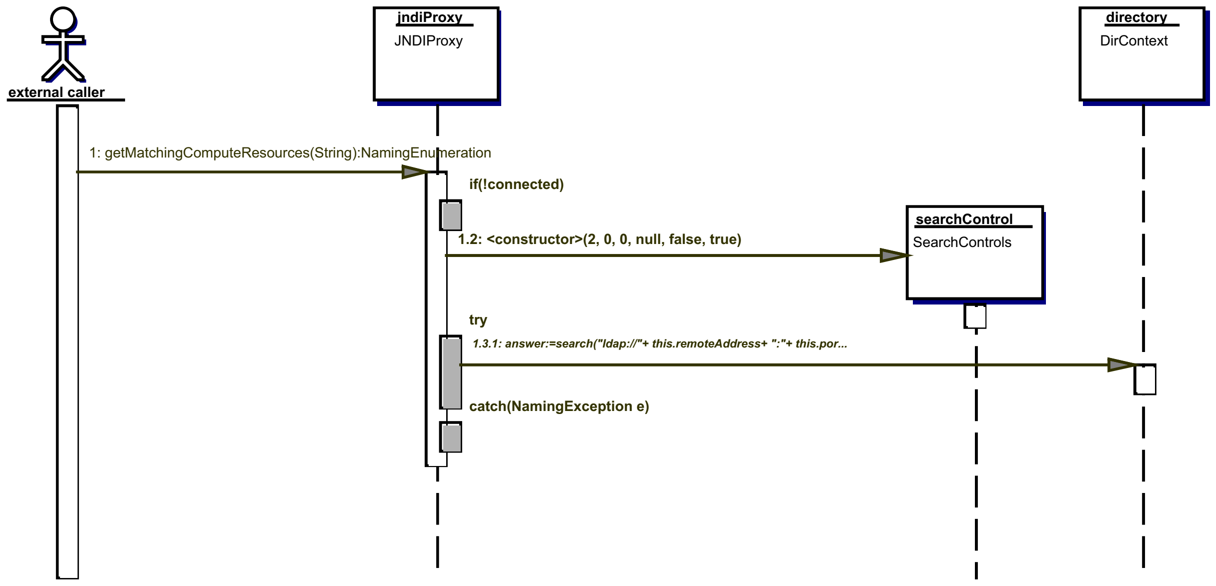

external caller
jndiProxy
JNDIProxy
directory
DirContext
1: getMatchingComputeResources(String):NamingEnumeration
if(!connected)
searchControl
SearchControls
1.2: <constructor>(2, 0, 0, null, false, true)
try
1.3.1: answer:=search("ldap://"+ this.remoteAddress+ ":"+ this.por...
catch(NamingException e)

Figure 3.38: Receiving matching resources.

3.6 Local (Sub-)Scheduler

In Grid Scheduling there are different parties with different interests. On the one hand there are those who submit jobs to be computed on the Grid. On the other hand there are owners of resources who want their resources to be available to the Grid. This availability, however, should not remain unrestricted. The provider of resources would want to remain in control of who uses them, when they will be used and what for they will be used.

GridSched features a component that is meant to address this issue. It was codenamed "Bob"

Bob allows 'physical' resources to take part in Grid Scheduling. This implies publishing, querying, reservation and administration of resources.

3.6.1 Requirements

Taking these considerations into account the following requirements for "Bob" arise.

- **Resources** In order to make resources available in the Grid, their existence needs to be published. Access to local resources is usually managed by a low-level scheduling system, perhaps partly integrated into the operating system. Resources that participate in Grid Scheduling must remain available for local users, too. Thus, the existing system scheduler cannot be replaced by a Grid Scheduler but a Grid Scheduler may be installed "on top" of it. In that case, reservation information must be propagated in two directions: From the Grid to the system scheduler and vice versa.
- **Resource Customization** Owners of large scale installations, like universities or companies usually organize their hardware in a hierarchical fashion.
- **Security considerations** While access to the resources of an institution may generally be permitted,access to selected departments of that institution may not. It is, thus, desirable to have hierarchical access control definitions.

3.6.2 Design

With regard to these requirements, Bob was designed as follows:
(Also see Figure: 3.6.3)

To manage resources that participate in Grid Scheduling, Bob features a data structure called ResourceManager. It stores references to each resource available to an instance of Bob.

Each so called Bob Resource features a datastructure called TimeTable. It is used to keep track of reservations that originated from the Grid and of reservations that were performed locally.

To synchronize those reservations, each resource that participates in Grid scheduling is assigned as SchedulerProxy that mediates between the ResourceManager and the resource's own system scheduler.

Access permissions to these resources are managed by the same policy manager that is employed elsewhere in GridSched (See Section: 3.9).

3.6.3 Implementation

For GridSched, one special kind of resource was realized: It consists of the service of running an executable on a computer system for a given span of time. Attributes of that resource are of course the properties of that computer system, like system architecture, physical memory, processor count and type, and so on. Scheduling and execution of processes is traditionally a task of the operating system.

The ResourceManager relies on a Hashtable that stores references to Bob Resources. The unique ID of each resource in GridSched serves as key.

The Timetable datastructure employs a TreeSet whose entries represent the beginning and the end of each reservation made for its associated resource. Reservations are propagated to the corresponding SchedulerProxy.

The SchedulerProxy starts and stops execution of an executable according to the beginning and ending of each reservation. Information on what to execute is contained within the resource description that comes along with the reservation. If execution fails, the corresponding reservation is automatically cancelled so that the remaining time can be used otherwise.

Resources of the ResourceManager are registered at a remote directory service via the ResourcePublisher class which employs GridSched JNDIProxy (See Section: 3.5) to interact with an LDAP Server.

Bob itself runs an endless loop which listens for incoming requests and takes action accordingly.

Example: Reserving Resources
The following is an example on how the process of reserving a resource typically looks like.

1. Bob is queried for time slots of a resource that was previously published by the ResourcePublisher. The ResourceManager then collects free intervals of time where the resource is not yet reserved.

2. Bob is then queried at least once for an offer on one of these time slots.

3. The resource is reserved as "grace".

   The ResourceManager creates an entry in the resource's timetable. After a predefined-defined grace period expires, this reservation will be automatically removed. (See Figure: 3)

4. The resource is finally reserved as "binding". This is done to confirm the grace reservation so that it will not be canceled automatically. The binding reservation will be propagated to the resource's SchedulerProxy which will notify the corresponding system scheduler of the reservation. (See Figure: 4)

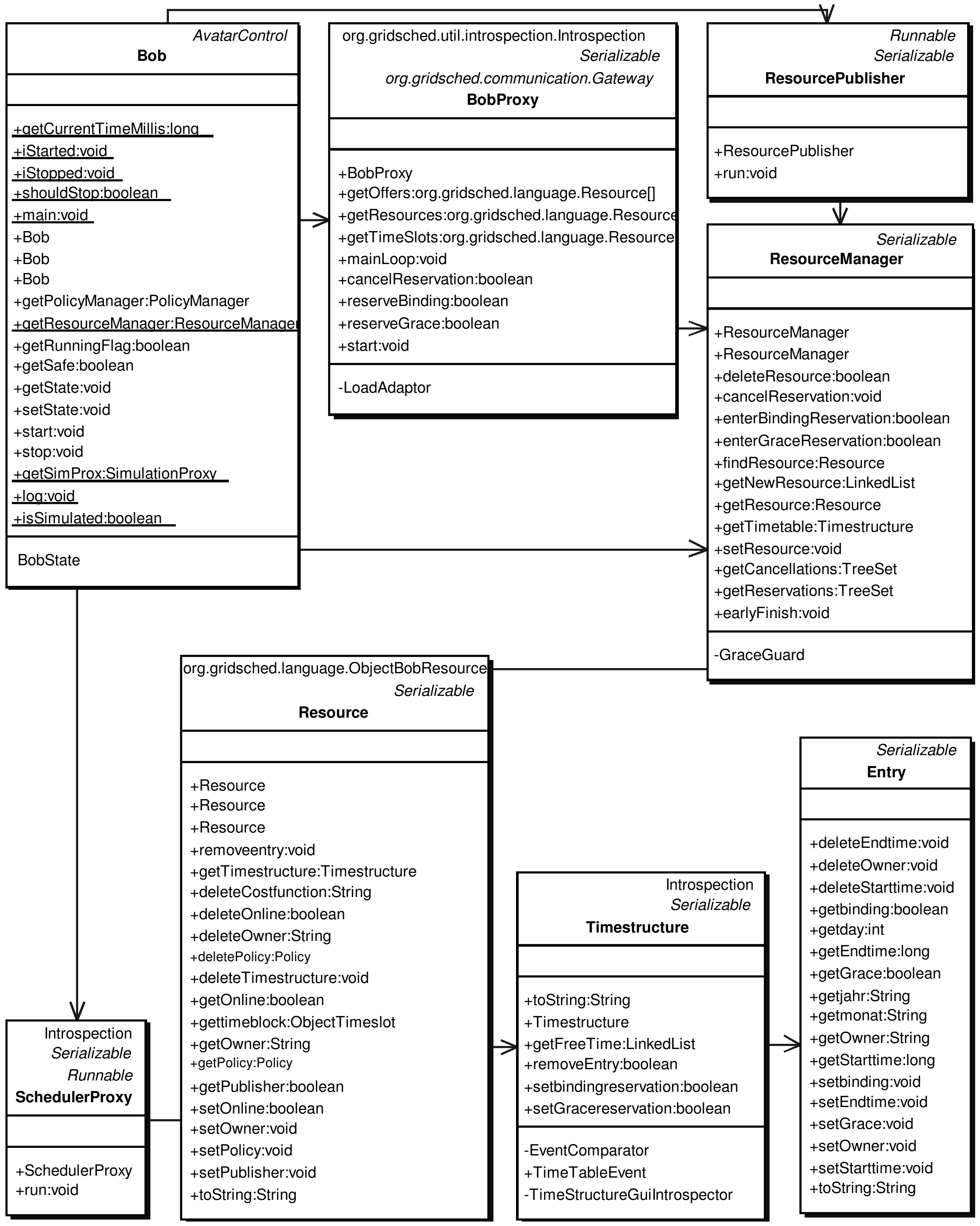


Figure 3.39: Bob Class Diagram.

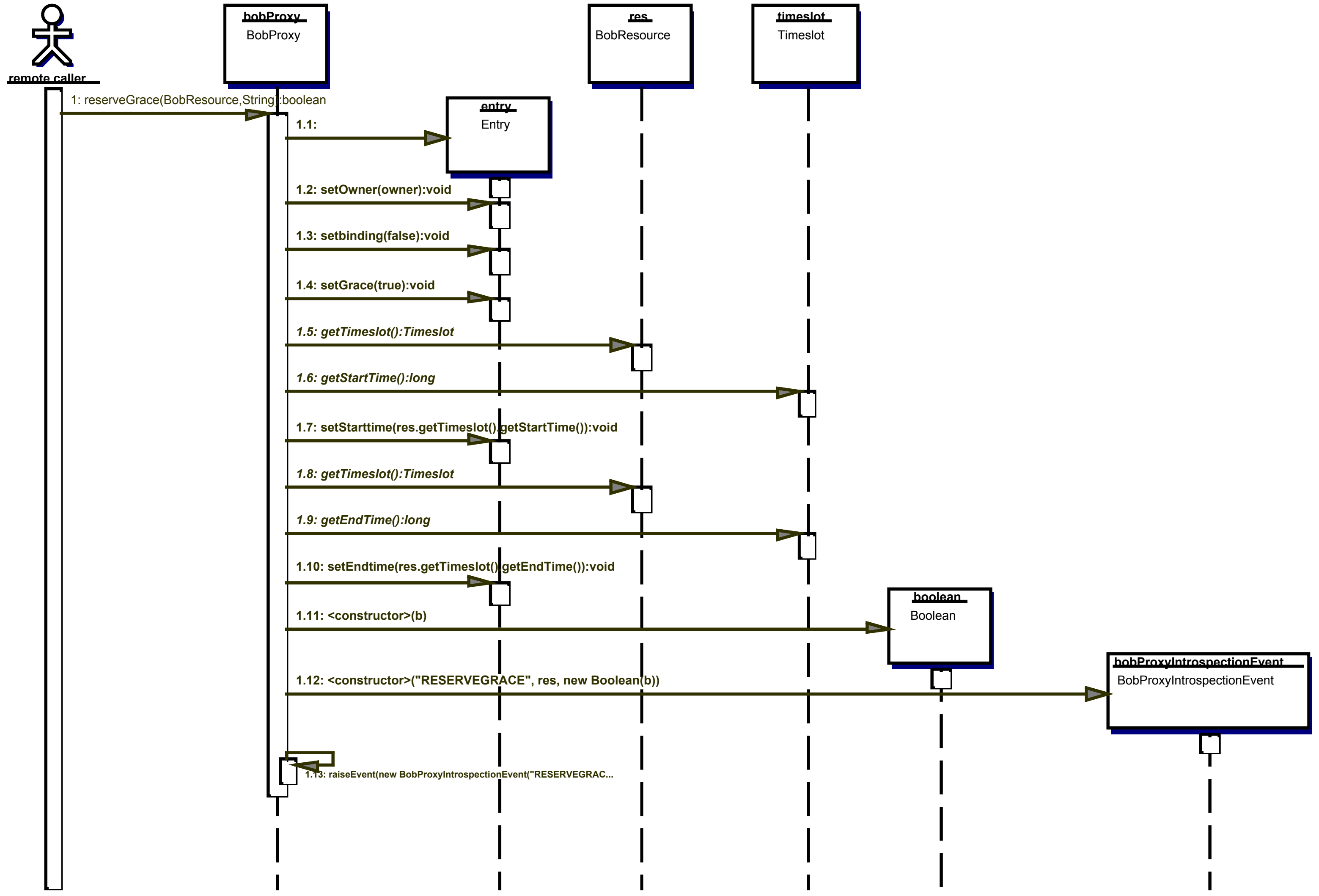

remote caller
bobProxy
BobProxy
res
BobResource
timeslot
Timeslot
entry
Entry
boolean
Boolean
bobProxyIntrospectionEvent
BobProxyIntrospectionEvent
1: reserveGrace(BobResource,String):boolean
1.1:
1.2: setOwner(owner):void
1.3: setbinding(false):void
1.4: setGrace(true):void
1.5: getTimeslot():Timeslot
1.6: getStartTime():long
1.7: setStarttime(res.getTimeslot().getStartTime()):void
1.8: getTimeslot():Timeslot
1.9: getEndTime():long
1.10: setEndtime(res.getTimeslot().getEndTime()):void
1.11: <constructor>(b)
1.12: <constructor>("RESERVEGRACE", res, new Boolean(b))
1.13: raiseEvent(new BobProxyIntrospectionEvent("RESERVEGRAC...

Figure 3.40: Bob GraceReservationDiagram.

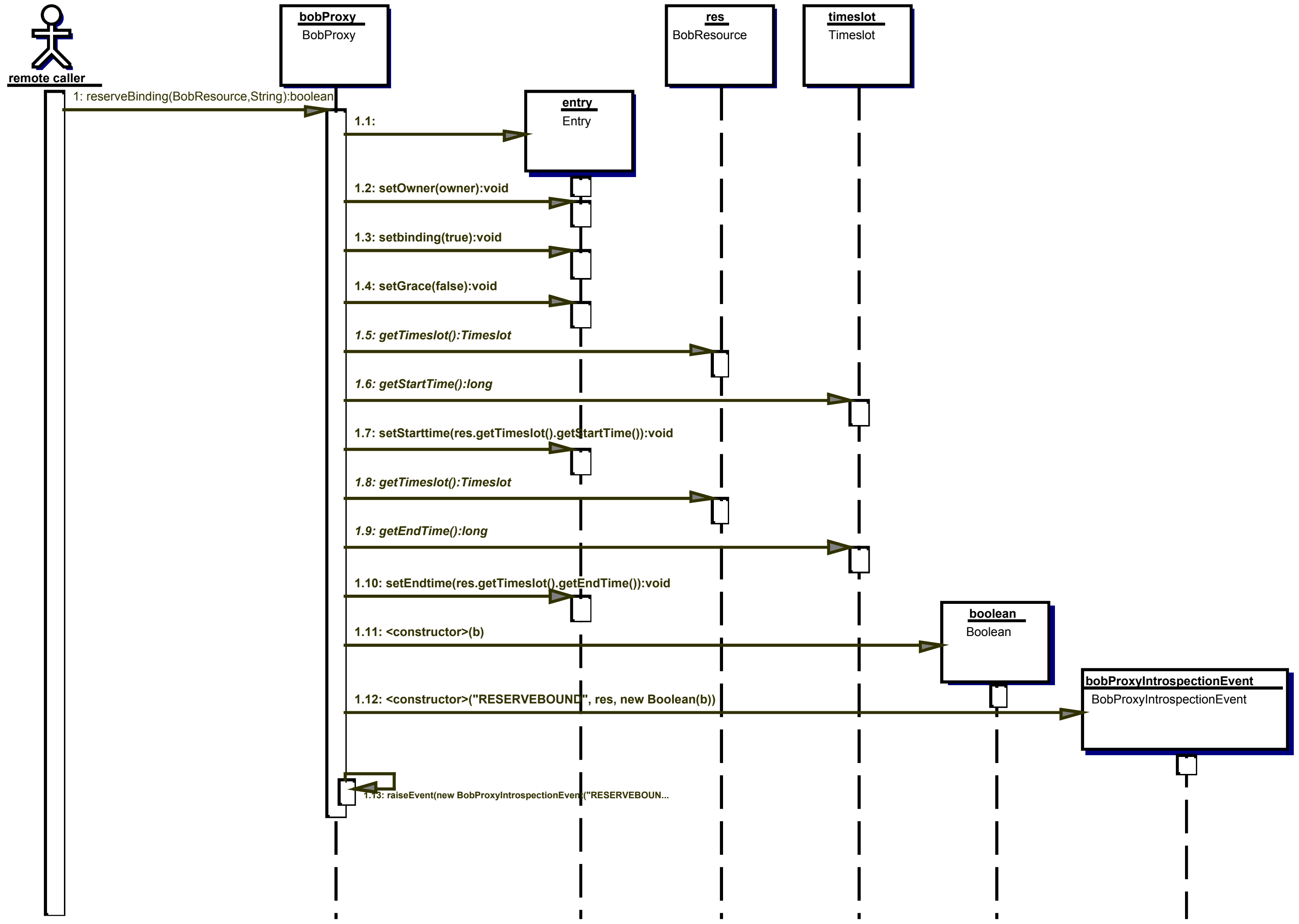

remote caller
bobProxy
BobProxy
entry
Entry
res
BobResource
timeslot
Timeslot
boolean
Boolean
bobProxyIntrospectionEvent
BobProxyIntrospectionEvent
1: reserveBinding(BobResource,String):boolean
1.1:
1.2: setOwner(owner):void
1.3: setbinding(true):void
1.4: setGrace(false):void
1.5: getTimeslot():Timeslot
1.6: getStartTime():long
1.7: setStarttime(res.getTimeslot().getStartTime()):void
1.8: getTimeslot():Timeslot
1.9: getEndTime():long
1.10: setEndtime(res.getTimeslot().getEndTime()):void
1.11: <constructor>(b)
1.12: <constructor>("RESERVEBOUND", res, new Boolean(b))
1.13: raiseEvent(new BobProxyIntrospectionEvent("RESERVEBOUN...

Figure 3.41: Bob BoundReservationDiagram.

3.6.4 Summary

- **Scheduler proxies** The current SchedulerProxy is capable to launch executables on the machine it runs on. Termination of a job, however, can become difficult due to the way how Java and the underlying OS handles child processes. No other kinds of resources are implemented.
- **Manipulation of resources** Resources can be added at initialization time by parsing information from an XML document. The same is done for policies.
- **Sub-Bobs** Originally it was intended to create the possibility of arranging Bobs in a hierarchical fashion. One Bob's resource manager would not only hold references to resources that it manages itself, but also to those resources managed by other instances of Bob which run on remote machines. Requests to such remote resources would be forwarded accordingly if the access permissions defined by the local policy manager apply. This way it would have been possible to have a hierarchical definition of resource access privileges.

3.7 DataManager

An important task combined with Grid Computing lies in finding and transferring data files. This is very important because files used in science and research tend to be very large [2]. So it is necessary to have special methods to transfer and locate files.

3.7.1 Requirements

It is necessary to have a replica service which knows where a file and copies of a file exist. Locating files is interesting because it might be that a file which should be transferred already exists on a computer which could be better reached, for example because of higher bandwidth [22]. This can help to reduce the time needed to transfer the file dramatically. Therefore it might be desirable to have a replica service which automatically distributes files in the Grid. This leads to the questions which files should exist, how often and where [9, 20].

Another important task is to manage the storage capacity on computers in the Grid which is used to store these replicas. On top of locating files it is necessary to transfer files from one computer in the Grid to another. There are a lot of different possibilities to transfer a file including partial file transfer or transfer from multiple hosts which should be supported. It should also be possible to remove files from the Grid which should not be used anymore.

On top of this the DataManager has to assure that special tasks can only be carried out by people which are allowed to do a task. Therefore it is necessary to have a service which authenticates users to the Grid and gives special rights to certain people or groups, e.g. it might be possible that a file should only be available to a certain group of users.

It is also necessary to reserve resources like files, network routes or disk space, so that a resource is really available at the time at which it is needed. Therefore it is necessary to have a kind of reservation service.

3.7.2 Idea

The DataManager component of GridSched tries to fulfill the tasks described above. The classes which were implemented to offer the desired functionality can be divided in several different parts.

- The control class `DataGate` which is responsible for coordinating all actions of the component. On top of this the `DataGate` class manages reservations, initializes all needed classes of the component and configures the settings to run it (see Figure 3.44).

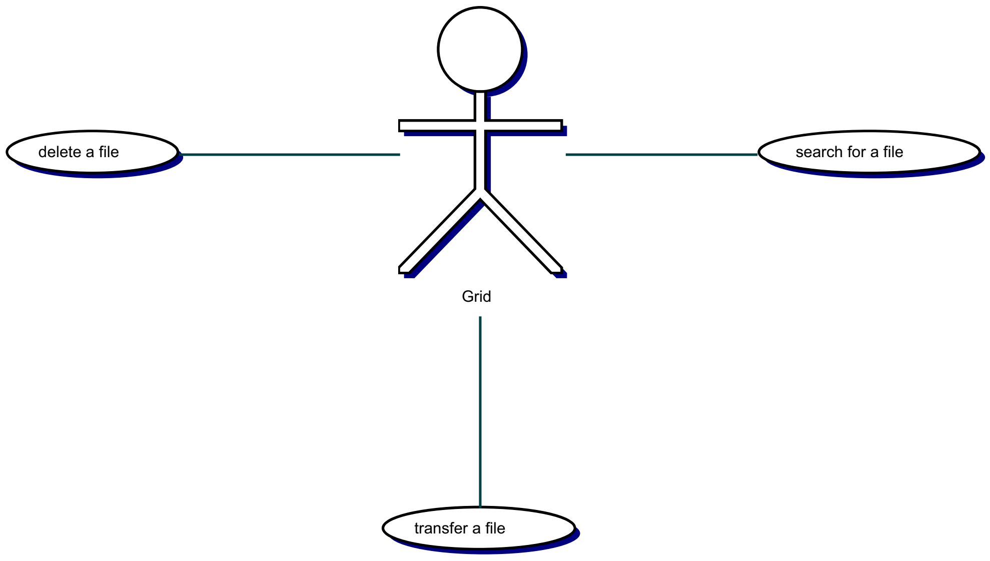


Figure 3.42: Requirements of the Grid towards the DataManager.

- Classes which manage the access to the database which is used to locate files. The database contains information about where a specified file exists or will exist. That means the database also stores information about future file transfers. For example if file x.dat should be transferred to computer y which will be completed at time z, the database stores that file x.dat will be available on computer y by the time z. So if a request arrives for file x.dat the component might offer to transfer the file to the wished destination from computer y. For more details see Section 3.7.3.3 and Figure 3.44.
  - `DbGate` - Serves to bind the component to the database. It is implemented to make it easier to use the database connection and to separate the rest of the program from the SQL syntax.
  - `DbConn` - Realizes the access to the database technically by using a special driver.
- Classes which administrate jobs. This is necessary because jobs do not need to be executed immediately. Therefore jobs which should be executed in the future are managed in a list until the job execution time is reached. A more detailed explanation is given in Section 3.7.3.4 and in the Figure 3.46.
  - `JobController` - Manages the jobs in a list and sorts them by the execution time.
  - `Job` - Upper class for all following sorts of jobs.
  - `JobAddToDb` - Extends Job and represents a job where a file should be added to the database.
  - `JobCopy` - Extends Job and represents a job where a file should be copied.
  - `JobDelete` - Extends Job and represents a job where a file should be deleted from the Grid.
- Classes which are responsible for the communication with other components. The DataManager acts like a server for other components of GridSched like Titan and NetworkManager. Therefore proxy classes exist which are listening for incoming requests from other components. After deciding what kind of request arrived the corresponding methods of the `DataGate` are called. Then it is the task of the DataManager to produce an answer for the request, e.g. to decide which file to use for a special transfer. In most cases it is necessary to question other components to produce a response for the incoming requests, for example network routes are reserved by the NetworkManager component. Therefore gate classes to these components exist.
  - `NetGate` - The class which sends messages to and receives messages from the NetworkManager component of GridSched (see Figure 3.45).

  - `BobGate` - The class which sends messages to and receives messages from the Bob component of GridSched (see Figure 3.45).
  - `DataProxy` - Listens for requests from other components (see Figure 3.44).

- A Class which manages the transfer of files. The `GassGate` which supports the transfer of a file between computers on the Grid. It was planned that the transfer should be done with support of Gass [10], a service of the Globus Tool Kit [8], but at least normal FTP is used. For more details see Section 3.7.3.6.
- The `Scheduler` class of the DataManager component creates a list of job bundles each with file and route for a request. That means the scheduler computes which file is most suitable for a transfer. The functionality is described in Section 3.7.3.5 and shown in Figure 3.45.
- Container classes
  - `SearchItem` is used as a container for all required information considering reservation and copying.
  - `File` which represents an entry of the database.

3.7.3 Implementation

In the following the implementation of the DataManager component is described. This includes the description of how the component is started and what happens during the running process. On top of this the database, the JobManager, the function of the scheduler and the file transfer are explained.

3.7.3.1 MainLoop/Start

There are two modes in which the DataManager component could be started. One is the normal mode, the other is simulation mode. There are a few differences between the two modes, e.g in simulation mode a file is not really transferred through the Grid. After the decision in which mode the component should run, the needed objects are created like

- the `Scheduler`, who makes decisions about which file will be transferred
- the `DbGate`, the connection to the database
- the `DataProxy`, responsible for receiving requests
- the `JobController`, who manages the creation and execution of jobs
- the `GassGate`, which is responsible for transferring files

During the startup of the component a configuration file is read and for example the location of the used database is configured. Another important task while starting up the component is to start the needed threads. There are two threads which are generated by the component

- the first thread is running in the `DataGate`, which is the control class of the component, it is responsible for checking if a job is ready for execution. Therefore the execution time of the next job from the `JobController` is compared with the actual time. The time used differs depending on the mode in which the component is running. In simulation mode the time is assigned from the simulation component while in normal mode the actual system time is used. If a job is ready for execution it gets executed.
- the second thread is running in the `DataProxy` which is listening for requests from other components. If a message arrives the corresponding methods to create an answer are called.

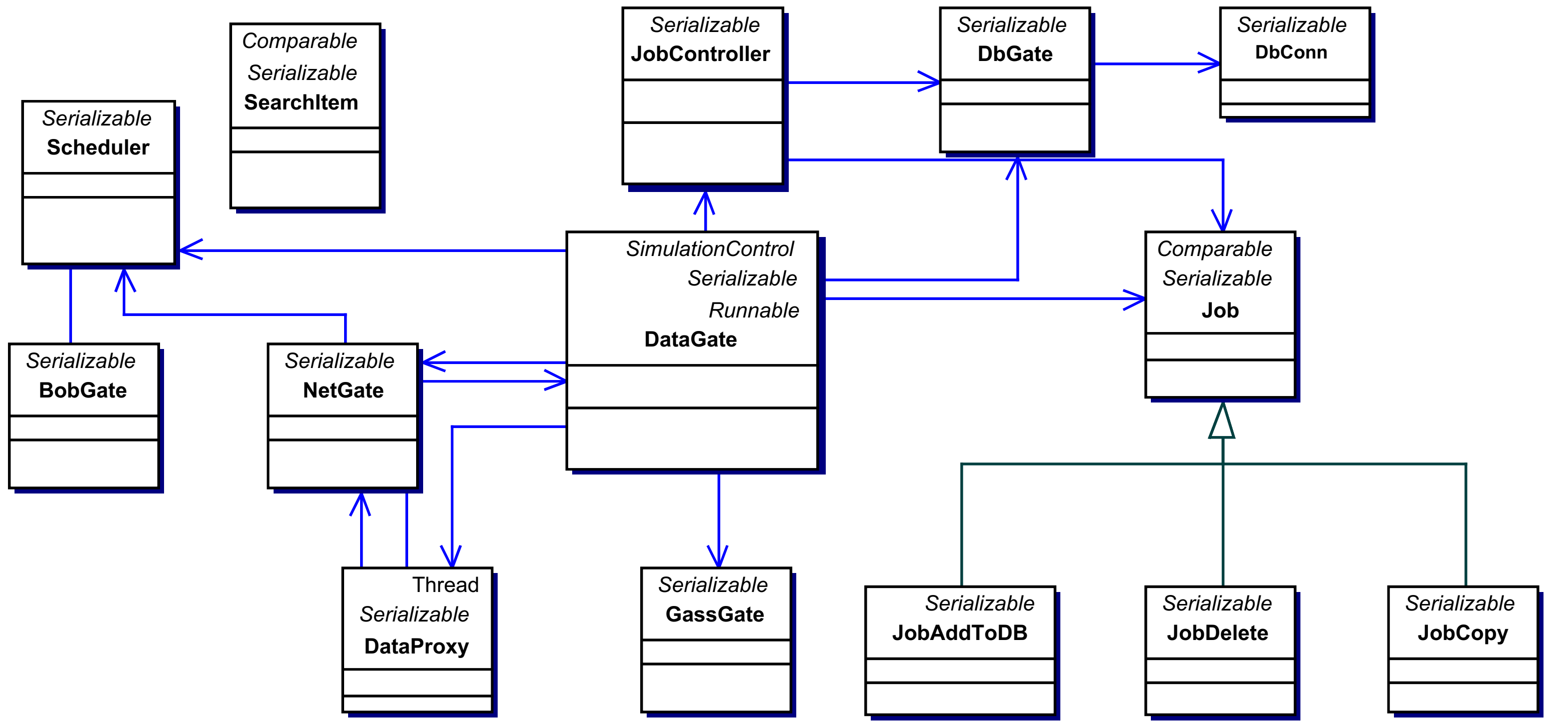


Figure 3.43: DataManager class diagram overview.

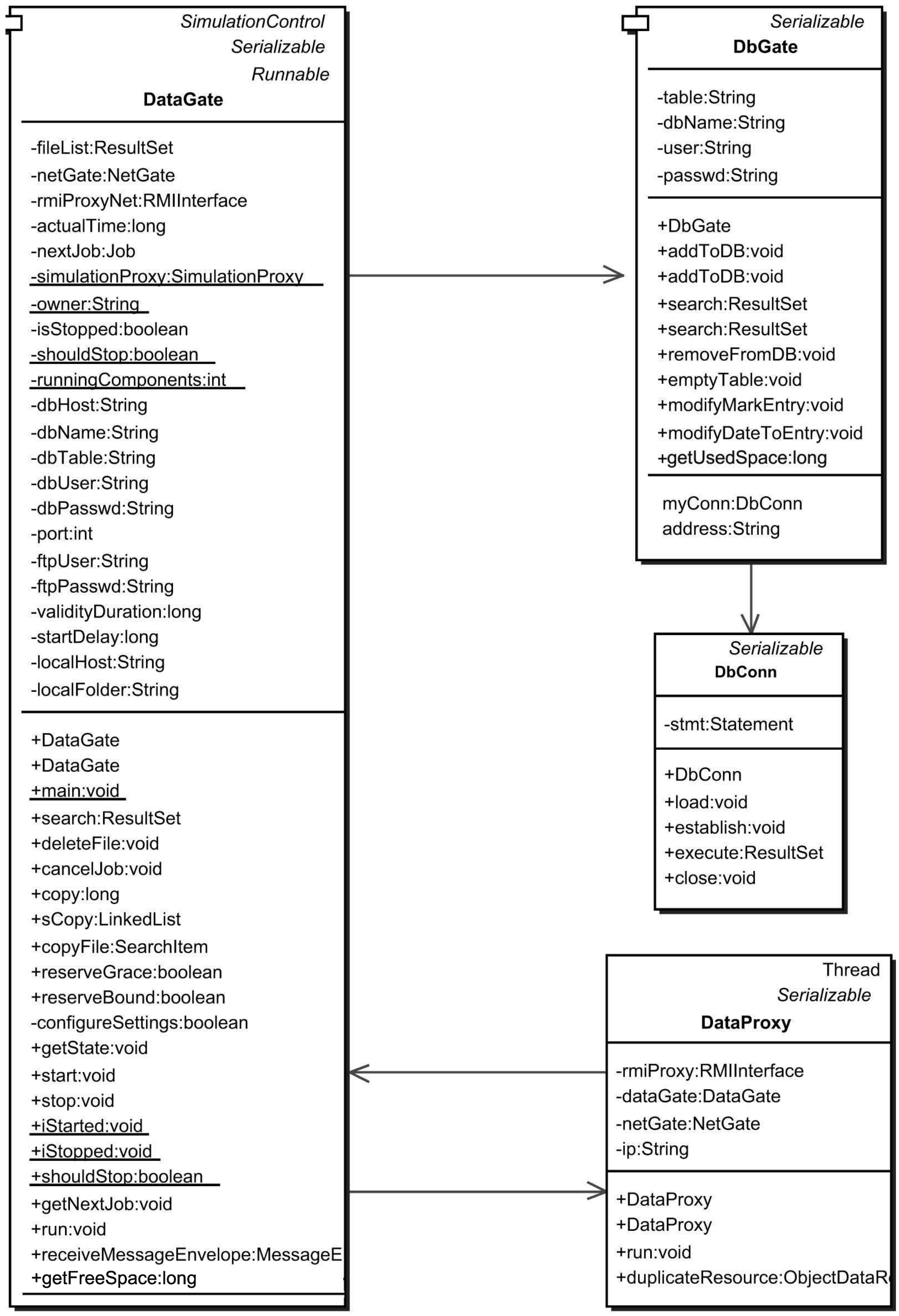


Figure 3.44: Control class and database classes.

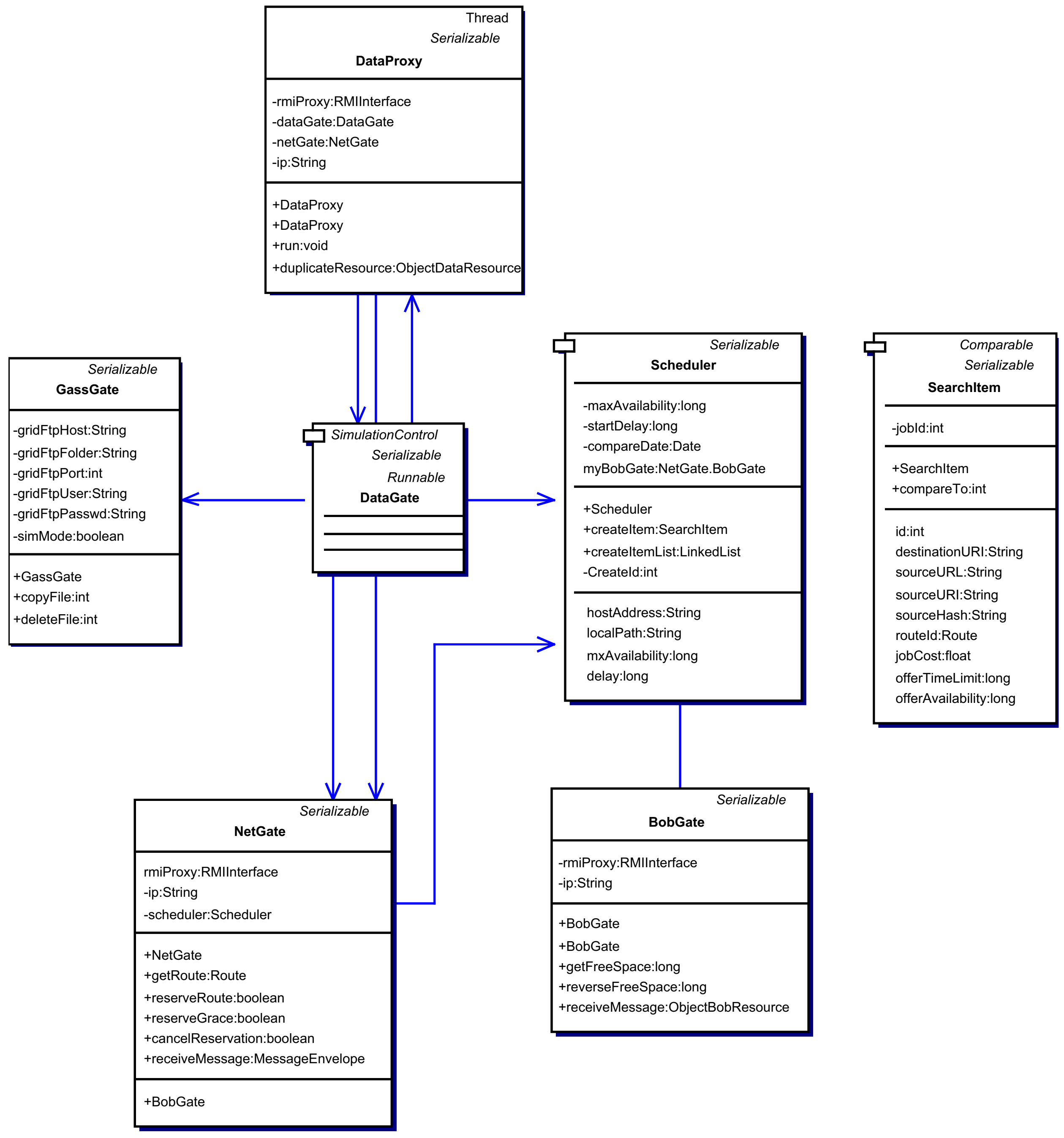


Figure 3.45: Scheduler class and gate classes to other components.

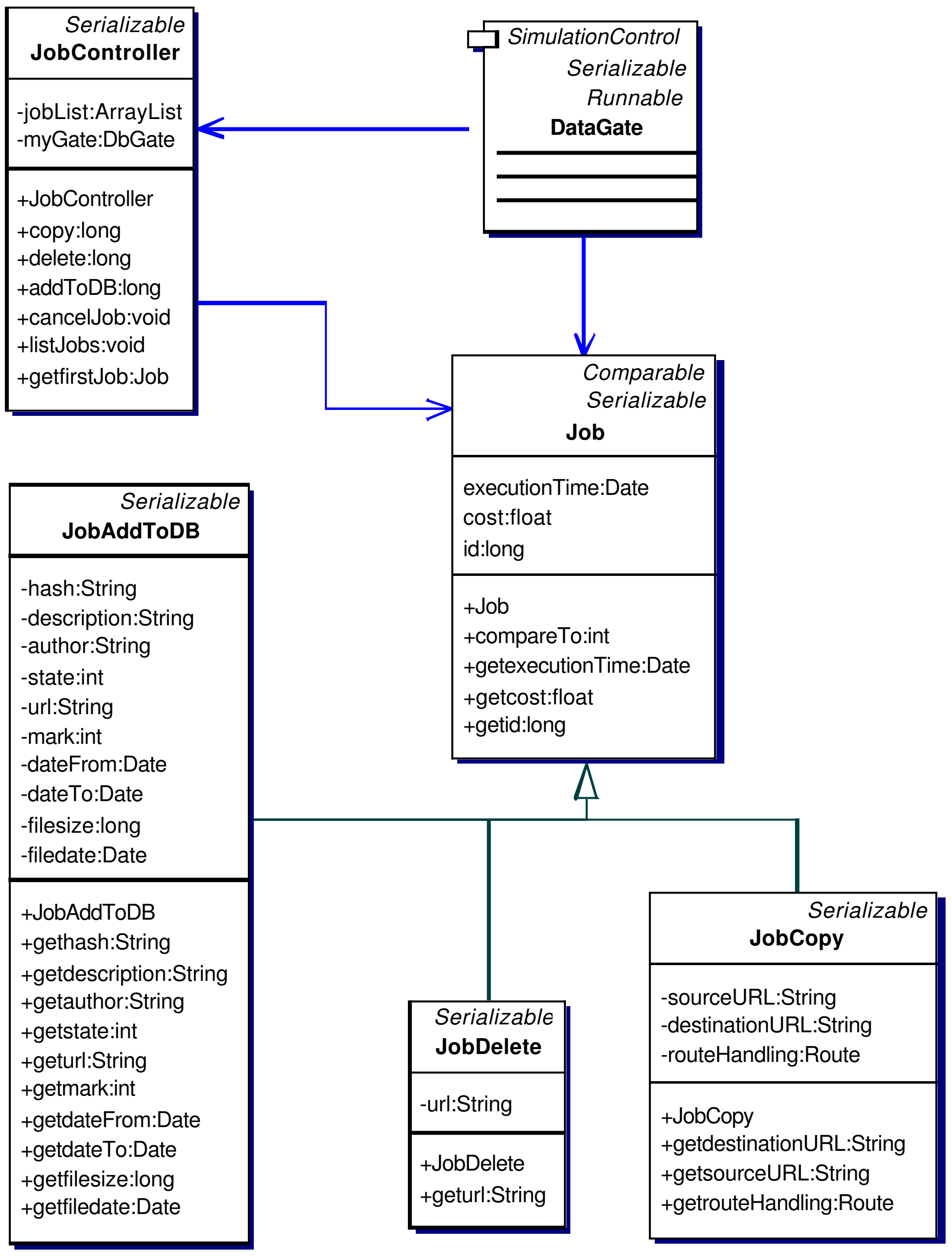


Figure 3.46: Job classes.

3.7.3.2 Communication

Like described above the `DataProxy` acts as a server for other components and is waiting for clients like Titan to send requests to the `DataManager`. There are several possible kinds of requests (see Figure 3.47):

- **SEARCH** - This request is send to find out if and where a specified file exists. Therefore the method `search()` of the DataGate class is called. Then it questions the database to find out where the specified file exists.
- **GETOFFERS** - Such a request is send to get an offer to transfer a specified file from a source to a destination. To create an answer the method `sCopy()` of the `DataGate` class is called. It first questions the database and then asks the scheduler to compute which file could be used most efficiently (see Figure 3.48).

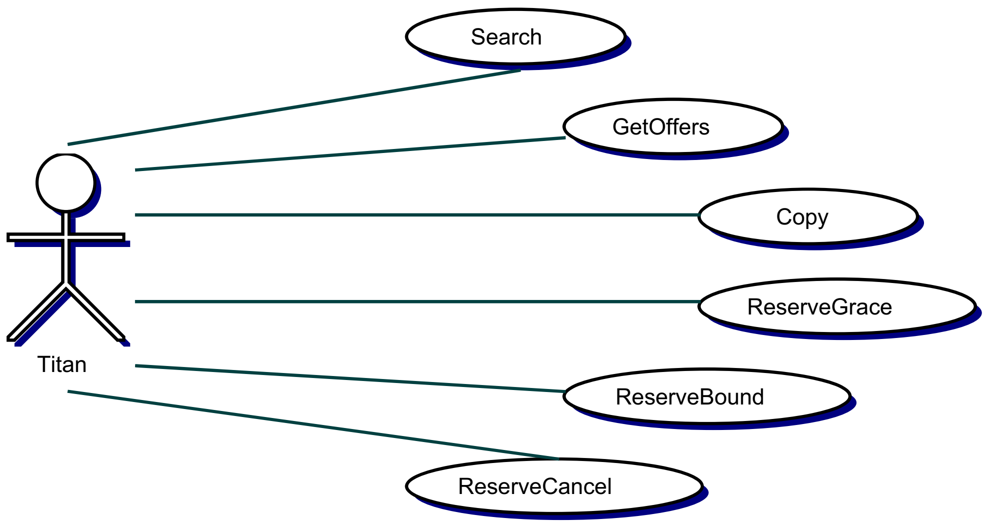


Figure 3.47: Possible requests from Titan.

- **COPY** - Copies a file from a source to a destination. Therefore the method `copy()` of the DataGate class is called. Then it directs the call to the `GassGate` to transfer the file.
- **RESERVEGRACE** - A grace reservation could be done after getting an offer and using the offer id. The method `reserveGrace()` of the `DataGate` class is called. Then it tries to reserve everything which is needed to transfer the file like the network route or the disk space needed to store the file on the destination computer.
- **RESERVEBOUND** - A bound reservation could be carried out after a grace reservation using the reservation id. To do so the method `reserveBound()` of the `DataGate` class is called. Then it tries to reserve the needed parts of the Grid.
- **RESERVECANCEL** - A reservation could be cancelled using the reservation id. If a reservation needs to be cancelled the involved components are informed to cancel a reservation.
- **FREESPACE** - This request is send to find out the free space of a given host at a specific time. Therefore a time slot must be given. The lowest free space in this interval is the result.

While the `DataProxy` is running in server mode and listening for requests from other components, the DataManager component also acts like a client of other components and uses their services to produce answers for incoming requests. For example to produce an answer for a GETOFFERS request from Titan it is necessary to ask the NetworkManager for a route between a source and a destination. In this case the NetworkManager component is the server and answers the request from the DataManager component. Therefore the NetGate class is used to send and receive messages to produce answers.

3.7.3.3 Database

The DataManager administrates a database in which the available files in the Grid are stored. To achieve a better availability and scalability the database is based on MySQL. The database serves as a catalog in which files are recorded. Search requests can send to the database which will return a result if a matching file exists.

Database Binding

The connection between Java and the MySQL database is provided by the library ConnectJ. The communication class `DbConn` realizes the most important part to establish a connection. Because of that the

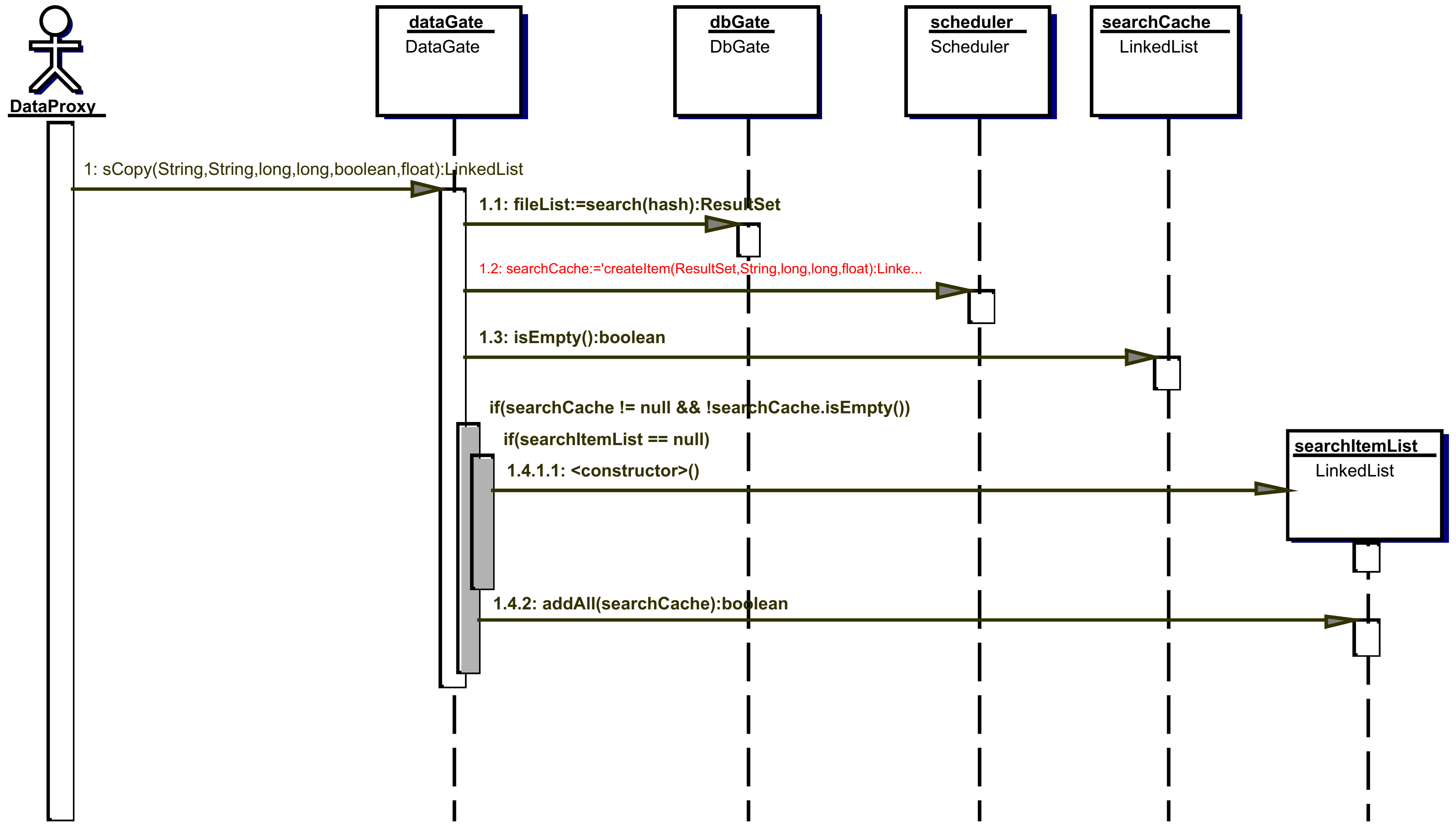

DataProxy
dataGate
DataGate
dbGate
DbGate
scheduler
Scheduler
searchCache
LinkedList
1: sCopy(String,String,long,long,boolean,float):LinkedList
1.1: fileList:=search(hash):ResultSet
1.2: searchCache:='createItem(ResultSet,String,long,long,float):Linke...
1.3: isEmpty():boolean
if(searchCache != null && !searchCache.isEmpty())
if(searchItemList == null)
1.4.1.1: <constructor>()
searchItemList
LinkedList
1.4.2: addAll(searchCache):boolean

Figure 3.48: Sequence Diagram: Search and copy a file.

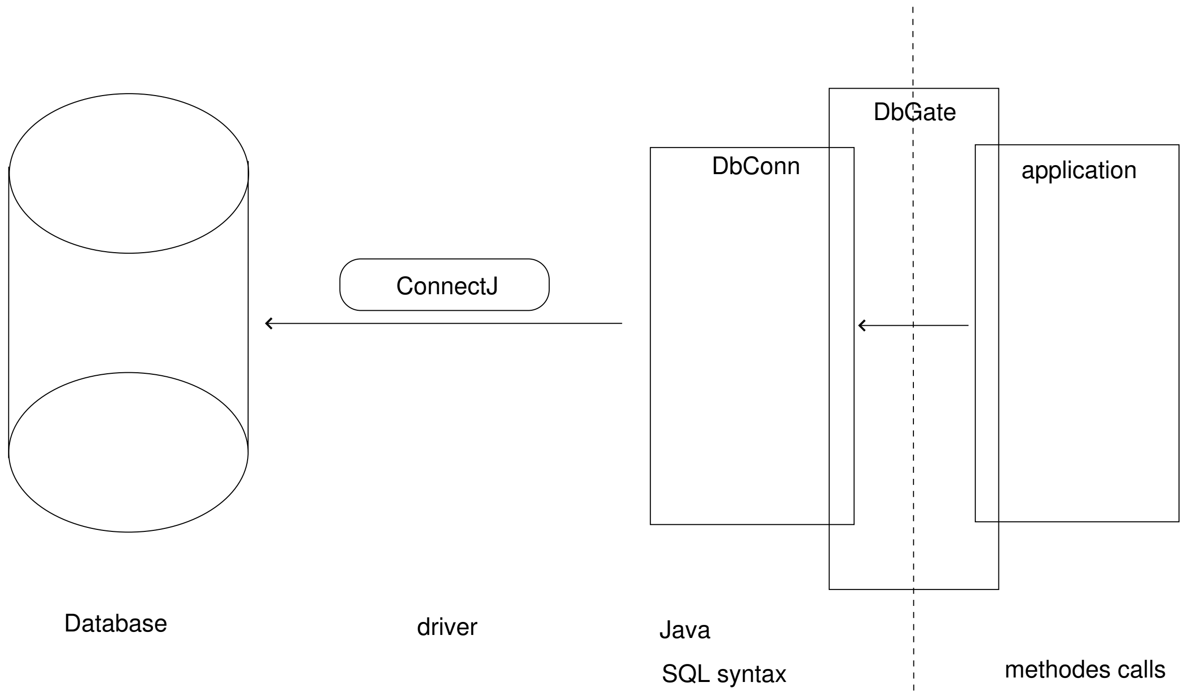


Figure 3.49: Scheme of the database.

classes which need to get information from the database do not need to care about further connection details. They just send a query to the database which then returns a result set that contains the answer to the query.

To establish a connection to the database some parameters are necessary like a user name, a password, the address and the name of the database. Then the ConnectJ driver can be loaded and a connection can be established.

The main job of the `DbConn` class is to receive a SQL query and give the produced answer of the database back to the calling component. This job is done by the `execute()` method of the `DbConn` class.

After using a connection, it must be closed by calling the `close()` method of the `DbConn` class.

Structure of the database

The key value of the database is the hash value which is the most significant part to identify a file in the Grid. The hash value is unique. It is generated by a secure random generator.

In the following the structure of the database is shown.

- hash - unique value to identify a file in the Grid
- description - contains additional information about the file
- author - the provider of the file
- state - gives information about the availability of the file
- cost - contains the price to work on a file
- URL - the resource locator of the file
- mark - if the bit is set, there is a job working on this file
- dateFrom - tells from which time the file will be available

- dateTo - the time until the file will be available
- filesize - contains the size of the file.
- filedate - the date the file was constructed

All important information and attributes of the registered files are readable from the database.

Using the database through DbGate

The class `DbGate` is implemented to make it easier to use the database connection and to separate the rest of the program from the SQL syntax. Every `DbGate` has its own connection to the database, which should increase performance when distributed databases are used. To initialize a `DbGate` it is necessary to set all parameters of a connection like user name, password, address, database name and table name.

The `DbGate` offers to add files to the database catalog with all required attributes. The required information is listed in the database structure. The class also offers the possibility to search for files in the database. Therefore it is necessary to pass a string to the `search()` method. The given string is then compared with the hash value, the description, the author and the URL. If matching records in the database could be found, a result set is given back.

On top of adding files to the database, files can also be removed from it. To execute a delete operation the URL of the file must be given.

As mentioned before the mark entry describes the availability of a file. The method `modifyMarkEntry()` can be used to modify the state of the mark bit which is set, when a planned job needs this file. When the job is executed the mark bit is reset.

3.7.3.4 Administration of jobs

The jobs the DataManager component is executing are jobs which are concerned with copying, deleting and adding files to the Grid. A job does not need to be executed immediately. A job contains an execution time and when this time is reached a job gets executed which is checked by the `DataGate` like described above.

The `JobController` class is responsible for managing the jobs. After a job is constructed it will be placed in a job queue which is controlled by this entity. Then it is possible to reorganize the jobs when a job got cancelled. The `JobController` also sorts the waiting jobs regarding the execution time.

Every `JobController` has its own gate to the database. This structure is used because of general database implementation schemes like described in the section before.

Jobs

There are three different kinds of jobs which are used (see also figure 3.46).

- `JobCopy` - Contains the information to copy a file
- `JobDelete` - Contains the information to delete a file from the grid
- `JobAddtoDb` - Such a job is executed to add a file which should be copied to the database, so that database also contains the new destination where the file will exist after the file transfer

All three kinds of jobs extend Job. The general job has an id, an execution date and costs for attributes. By initializing a new job the id is set automatically. The id is also the return value when creating a job. The Job implements the interface comparable. So jobs can be sorted in the order of execution.

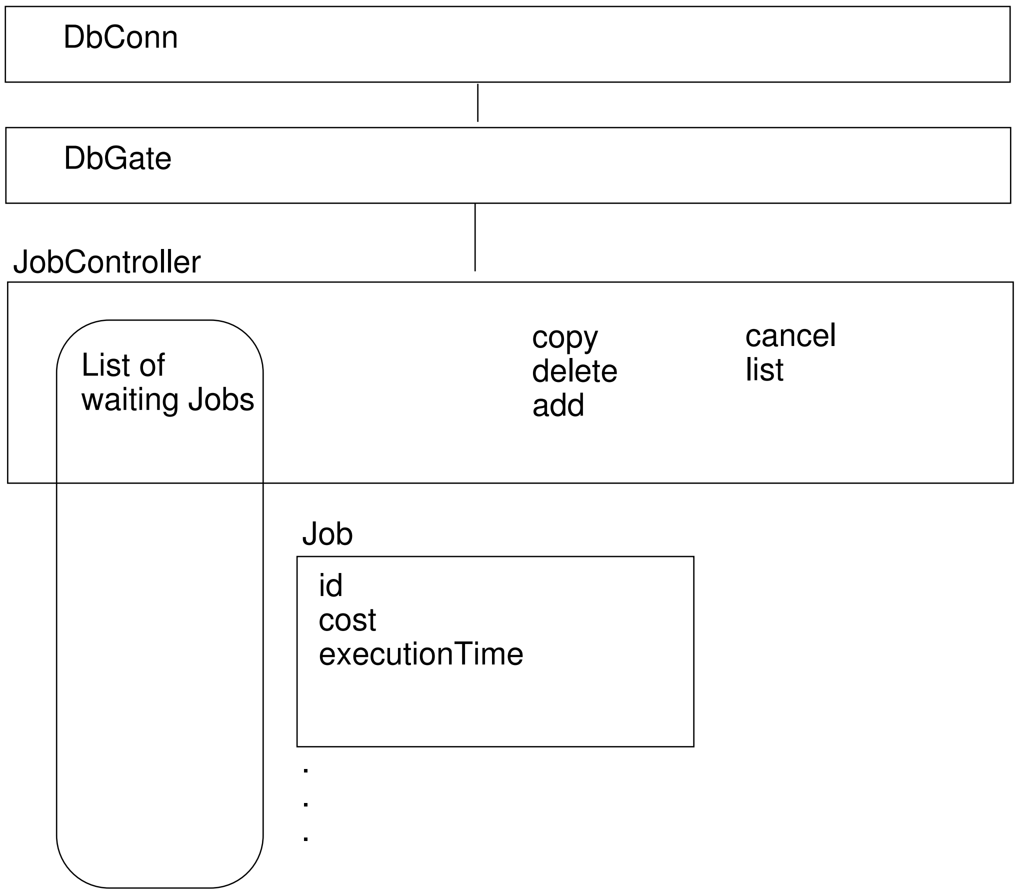


Figure 3.50: Scheme of the job controller.

JobCopy

To copy a file information like the source, the destination and a network route is required. This parameters must be given to create a copy job.

If a new copy job is created some actions need to be done to make further operations efficient. First the source file must be marked because it is not allowed to delete the file before the job is executed. Second it is necessary to add the destination file - even if physically not yet present - to the database. The dateFrom field is set to the execution time of this job. So the scheduler knows with the help of the database that there will be a copy of a file at the given time in the future. Before a CopyJob is accepted the free space of the destination host is checked. If there is not enough space, the job is refused.

When cancelling a copy job, the source file must be unmarked, and the destination file must be deleted from the database.

JobDelete

To delete a file it is just necessary to specify the URL of the file. The file will be marked and the dateTo field is set to the execution time of this job.

In case of cancelling this deletion job, this file will be unmarked and the dateTo field is set to null.

JobAddToDb

To add a file to the database a `JobAddToDb` is required. All parameters as described in the structure of the database are necessary. At the given time the file will be available in the grid.

When such a job is cancelled, only the job object will be removed.

3.7.3.5 Scheduling

The scheduler is activated by the MainLoop each time it is working on a GETOFFERS request.There are two ways of scheduling: scheduling with external and scheduling with internal files. In both cases the scheduler gets fixed defaults like how long the file will be needed, when the transfer must be completed, the maximum amount which the transfer is allowed to cost and the time from which on the file be available. The availability is especially important if the result of one job is the basis for the next job.

Scheduling with an internal file

If a file on the internal host is required for a job, the MainLoop starts the schedule method `CreateItemList()`. For this list a result set of the MySQL database is required which contains all potential copies of the needed file. Potential means that it is possible that the result set contains files which are not physically existing yet, but are planned to be copied to that destination. The result set is provided by the MainLoop.

First the scheduler checks the defaults, e.g. if enough storage capacity on the destination host is available and how long. After this the scheduler checks the result set and picks out all files who can't fulfill the defaults because they don't have enough capacity or are too expensive. The remaining files will be checked to find the quickest and cheapest available transportation route.

For all bundles of a file and route which fulfill the defaults a `SearchItem` will be created. The `SearchItem` is used as a container for all required information considering reservation and copying.

All `SearchItems` will then be sorted regarding their costs and will then given back to the MainLoop in a list.

Scheduling with an external file

If a certain file of an external host should be copied for a job the MainLoop uses the scheduling method `createItem()`(see also Figure 3.51). The scheduler gets the common defaults and apart from that special information about the file like the name, availability, address and maximum size. If the file is the result of a job which is not finished yet its size is not known then of course. In those cases it is necessary to work with assessed (maximum) values. The further course is exactly the same as in the method `createItemList()`. As a result this method gives a single `SearchItem`.

The SearchItem container

For the reservation of all needed components and resources different information are required. They are gathered in the `SearchItem` class (see Figure 3.45). For sorting all `SearchItems` of a list the method `compareTo` is used which sorts the `SearchItems` by cost.

The `SearchItem` container contains the following information:

- **jobId** unique id for all `SearchItems` for a request
- **jobCost** whole costs for all needed resources
- **sourceURI** full address of the source file
- **destinationURI** full address of the destination file
- **sourceHash** unique string used for file-identification
- **routeId** needed for the reservation of the transfer route
- **offerAvailability** end-time for the validity of the bundle-information

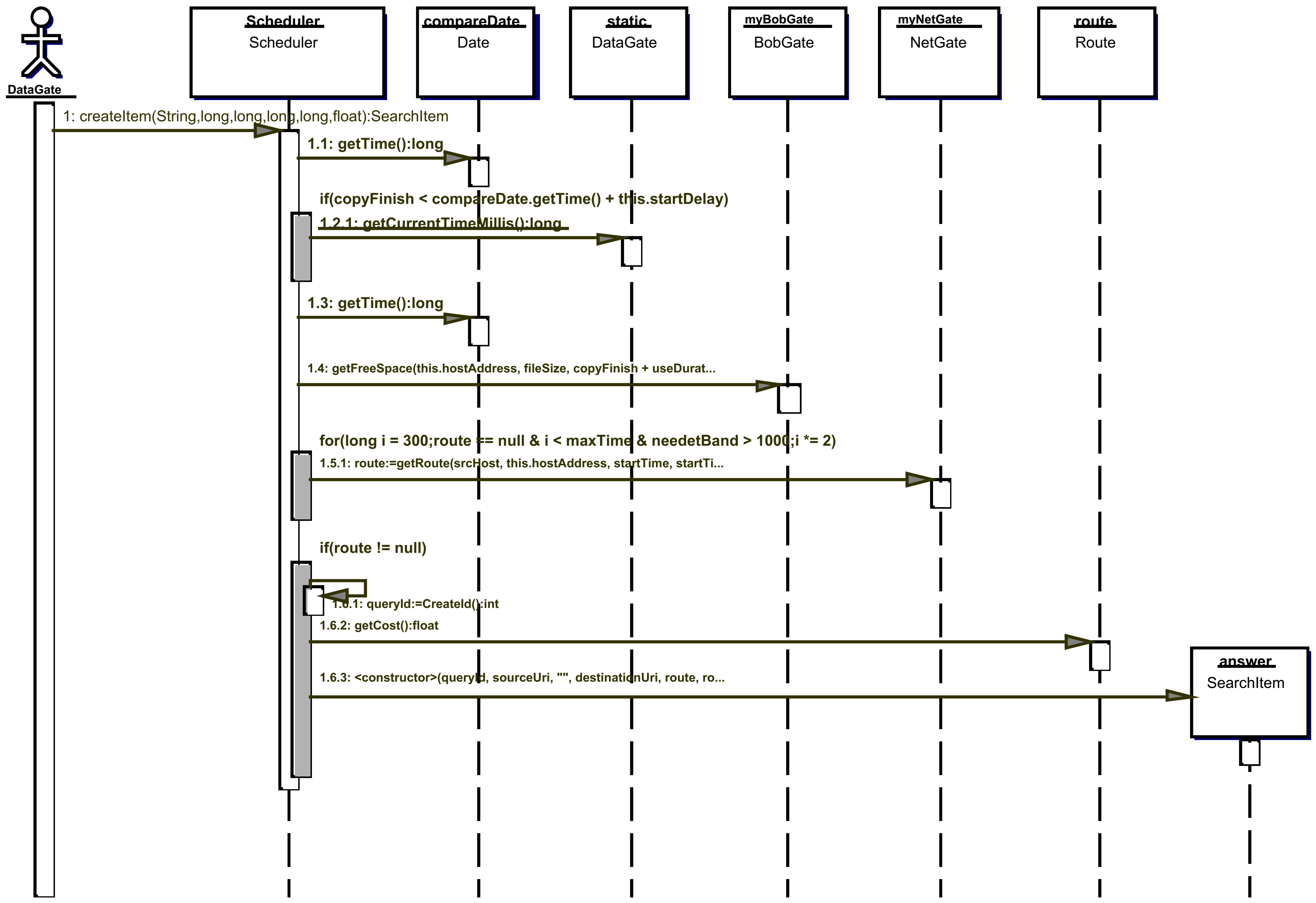

DataGate
Scheduler
Scheduler
compareDate
Date
static
DataGate
myBobGate
BobGate
myNetGate
NetGate
route
Route
answer
SearchItem
1: createItem(String,long,long,long,long,float):SearchItem
1.1: getTime():long
if(copyFinish < compareDate.getTime() + this.startDelay)
1.2.1: getCurrentTimeMillis():long
1.3: getTime():long
1.4: getFreeSpace(this.hostAddress, fileSize, copyFinish + useDurat...
for(long i = 300;route == null & i < maxTime & needetBand > 1000;i *= 2)
1.5.1: route:=getRoute(srcHost, this.hostAddress, startTime, startTi...
if(route != null)
1.6.1: queryId:=CreateId():int
1.6.2: getCost():float
1.6.3: <constructor>(queryId, sourceUri, "", destinationUri, route, ro...

Figure 3.51: Sequence Diagram: Scheduler.createItem().

Classes

The following classes are needed for the scheduling-process:

- **org.gridsched.dataManager.Scheduler** Creates a list of job bundles each with file and route for a request. The primitive algorithm works on a small number of files. Many files can urge a very long runtime.
- **org.gridsched.dataManager.SearchItem** Container class for a job bundle with file and route for a request.

3.7.3.6 File transfer

At the moment `GassGate` uses simple FTP to transfer a file. You need a FTP-Server for copying the files. Locally `GassGate` uses a simple FTP-Client.

The aim is using tools based on the Globus toolkit 3, for example GridFTP [21, 20] or GASS (Global Access to Secondary Storage) [10]. Both GridFTP and GASS use the GSI (Grid Security Infrastructure) for authentication and secure data transfer. Thus, you will need to acquire the GSI credentials before you can transfer any data.

Copy a file

The method `copyFile()` needs the address of the source file, the address of the destination file and a route for the transfer. At the moment the route will be ignored.

Delete a file

You can only delete a local file. The `GassGate` method `deleteFile()` needs the address (URL) of the file. It uses the class „java.io.File".

Classes

The following classes are needed for the file transfer:

- **org.gridsched.dataManager.GassGate** Interface to Globus Gass service. It copies or deletes files.
- **org.gridsched.util.ftp.FtpClient** Simple implementation of a FTP-Client.

3.7.4 Summary

There are some features which could be added to the DataManager, but these are only things, that are not really necessary.

- **General** There is no automatic replica service included jet. This service could start transfers by itself to distribute the files in the Grid.
- **Database** The database could be improved by using a distributed database to make queries more efficient and to assure a good availability of the database.
- **File transfer** A change to GridFTP or GASS with certificates would offer different new possibilities like partial file transfer or different transfer options for different users. But it is decided against the Globus Tool Kit and so this feature is also gone.

3.8 Network Management

The grid consists of computing machines connected via a network. It could also be seen as a graph consisting of nodes and edges, whereby the nodes represent the machines that offer resources like data or storage and the edges represent the connections between them.

The network is just another resource in the grid, offering a means of communication and transportation. Therefore it is obvious that a management component for this resource is in need.

The NetworkManager in GridSched is the central component concerning network management including the reservation of certain network routes and offering certain network routes with regards to attributes like bandwidth or costs.

3.8.1 Motivation

Concerning grid scheduling, different requirements occur to the system. It may be necessary for a calculation job, to transfer great amount of data from one place to another respecting strict time constraints, so that the job can be calculated on time. It may also be necessary to support a minimum promised bandwidth for telephone and video conferences. So problems concerning data management always involve problems concerning network management. It is not only sufficient to know what data is located at a specific node at a special time, but to be able to transport this data to another location in a specific amount of time. Looking at the grid, it consists of nodes which represent different locations within the grid. These locations are computers which have a unique address. To copy data from one place to another, it is necessary to know the structure of the grid and the states of single attributes like bandwidth, latency, throughput and so on.

To transport data from one place to another it is necessary to reserve a route through the grid and to be able to make sure that special attributes of the route can be assured. Therefore it is ideally optimal to know every single component of the network and to be able to report the state of every single attribute at a special time. So the scheduler can always find and reserve the "optimal" route within the grid. It is therefore necessary to maintain a monitoring and reserving mechanism. The global grid is a structure that is not limited by state or enterprise borders. It is composed of many nodes in different sub-nets that belong to different owners. Necessarily it must be possible to calculate costs for reserving and offering routes. So a cost component is necessary. All previously mentioned functions find themselves implemented in the NetworkManager.

3.8.2 Requirements

The main duty of GridSched's NetworkManager is to provide a network connection to move data from one node of a network to another. This particular network connection may be requested by GridSched's DataManager or directly by the scheduler. Both nodes, the starting and the ending node, of a network connection may be assigned to different network domains.

Network connections are often needed for a special period of time which may be nested in a specified time interval. Moreover, a requested network connection has to fit to some quality aspects, i.e. bandwidth and latency which are also known as Quality of Service (QoS). In order to achieve certain levels of QoS, it is necessary to reserve a specific network route which represents the requested network connection. With this step, it is prevented that other requests for a network route may cause conflicts which compromise the requested QoS of a reserved route. The fact, that some network connections cannot be reserved, is also regarded in NetworkManager´s approach.

Another critical aspect of network route reservation is its monetary price. The usage of a reserved network route causes certain costs to its owner or provider which are billed to its user. Therefore, costs of connectivity of all participated nodes must be stored in a central database of GridSched's service.

3.8.3 New Idea / Major Changes to previous idea

In the new approach to the solution of our problems, it was generally abstracted from the technical mechanisms of reservation and focused on the administration of the reservations . Therefore, GARA has vanished from the management of our network. In fact, we have never been able to test our environment with using GARA. Several computers would be needed to construct a testbed that is capable of outputting relevant information and results.
Several internal discussions and one discussion with the inventor of GARA made us think about our way of using NWS as well. Until then, NWS has been used as a forecast service for determining the bandwidth of routes at a given time.
In the new approach the network grid built of two graphs, a local and a global graph, was considered(see Figure 3.52).

3.8.3.1 Global Graph

The global graph represents the network considering network domains. These domains can be seen as a heap of network nodes geographically close to one another. E.g. the computers of the university of Dortmund represent such a domain. Every domain is represented by one NetworkManager, the network component of GridSched. That means for every one domain, a NetworkManager has to be installed and run. Every single domain is represented by a node in the global graph, and an edge in the global graph represents a direct connection between such two domains.

This graph will obviously be highly fragmented, because a direct connection between every two Network-Managers is highly unlikely.
But it could also be possible to add edges to the graph that do not represent a direct connection. With the help of a metering tool it could be possible to determine e. g. bandwidth and other important attributes regarding the connection between two nodes. So it could be assumed what the connection does look like and an edge could be added to the global graph representing this connection.

3.8.3.2 Local Graph

If the global graph is built of nodes which represent network domains, the local graph is built of nodes that stand for the machines within the domains. Taking the former example, the local graph would represent the machines within the university of Dortmund.

The local graph represents a network considering real existing nodes in between a NetworkManager's domain. Because all participating nodes are under the control of one Administrator and very near to each other, the resulting graph should not contain isolated nodes.

3.8.3.3 Reservation

As mentioned before, our new approach to solve the problem regarding reservation abstracted from the technical reservation itself. To be able to reserve network connections using GARA, you have to own or at least know every node on the route you are trying to reserve and on every node a special service has to be installed and run. That makes it very hard to reserve routes between NetworkManagers, because it is highly unlikely that a direct connection exists. Most of the routing will touch unknown territory. That and the problem, that a mechanism to send the data over the former reserved route would be needed(some kind of routing protocol), induced the decision to remove GARA from the former approach and take one step back.
Now Reservation is being made by using an universal reservation component that is capable of reserving network routes considering former reservations regarding attributes of those reservations such as bandwidth and timeslots. So reservation is being taken into consideration, the reservation of real machines could not be accomplished.

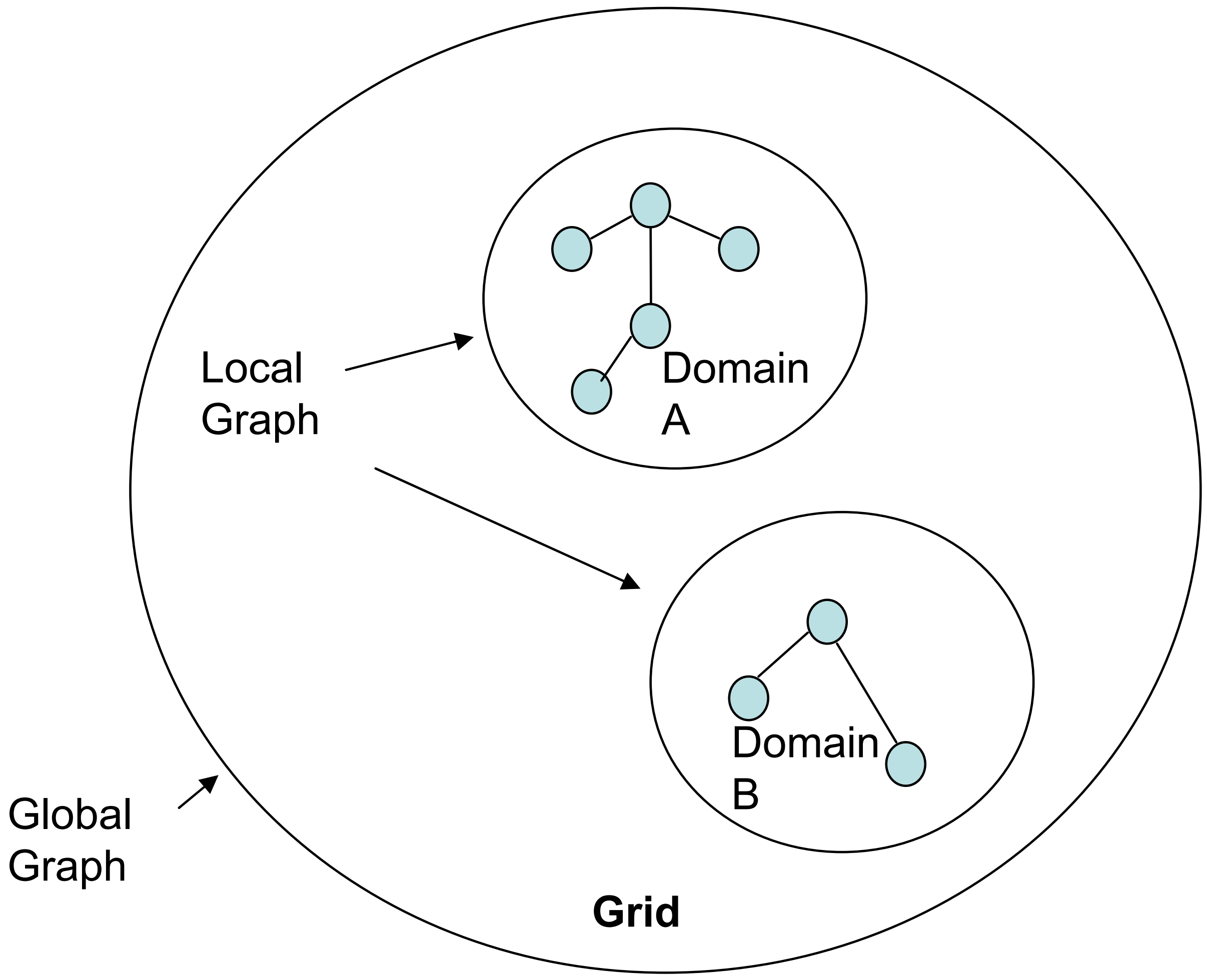


Figure 3.52: Global and Local Graph.

3.8.3.4 Forecast

When the scheduler or DataManager in GridSched needs network connections for transferring data from one place to another, it places a request to the NetworkManager. The request contains information about the quality parameter of the needed connection (e. g. bandwidth) and the timeslot(s) that refer to the start- and endtime the connection is needed.

If the NetworkManager has to be capable of presenting valid and precise offers, it has to be aware of the state the network will be in at the time the connection is needed. To be able to make such predictions, a forecast service must be at hand that meters the routes that will be offered every time of the day continuously. Such a service also has to be aware of other reservations that are being made in the network. E. g., if a route with a given bandwidth of 100 Mbit/s has been reserved at 18:00 hours with a bandwidth of 60 Mbit/s and a NetworkManager, not aware of that, offers this route at 18:00 hours with a given band- width of 70 Mbit/s, the parameters for one or both of the reservations cannot stand.
So, to implement a forecast service, the reservations being made in the network have to be saved. It is also necessary to meter any routes to be able to predict situations for making precise offers. The first point could be implemented very easily, but to implement the second point in a well-formed manner, every route in the network that is managed has to be known. The state of the network must be known at every time. This cannot be done easily if the network that is managed is as big as the one GridSched is working on. First, not every route that exists is known. You may be able to manage small networks like

the ones the former mentioned local graphs refer to. But taking a network our former mentioned global graph consists of, only a few direct connections exist between two domains that are well known and thus can be managed. The number of unknown connections is extremely high and therefore unmanageable for neither a reservation nor forecast service. Even if an edge is added to the global graph that stands for the network between the two existing nodes, the state of this network can not be predicted precisely , because the route metered before has not to be taken the next time this network is metered.
These problems lead to the decision to rather work on other problems concerning the network component than solve this problem. Therefore a forecast service does not exist in GridSched's NetworkManager.

3.8.4 Implementation

The NetworkManager's implementation is being divided into 3 packages: main, visibleGraph and graph. The main package provides the main functionality of NetworkManager. The central class is the `NetGate` class that responds to all requests that are coming from other components. The constructor reads a properties-file called "nmConf.xml" that contains information about where to find the local graph and the LDAP server. Using the main method within NetGate, all necessary components relevant to the NetworkManager are started properly. As the central interface to the "outside world" `NetGate` contains the following methods:

- getOffer()
- reserveGrace()
- reserveBind()

The `NetProxy` class deals with incoming requests using RMI as the means of communication from the NetworkManager to other components. It can be used by a given set of requests: "getOffer", "reserveGrace" and "reserveBind".

If a request deals with several NetworkManagers, not only with the one getting the request, the `NetToNetProxy` class splits the request into pieces that concern the other NetworkManagers and sends new requests to them, so that an offer or reservation can be generated properly. These newly generated requests are called "getInternalOffer" and correspond with the `Reservation Manager` from the domain the request is sent to. The Reservation Manager calls the method "findPath" from its `LocalGraph` class that returns a route through the local domain.
The `Route` class is still the class that represents a network route concerning the main attributes bandwidth and start- and endpoint.
The `LDAPObject` and `LdapGate` classes are used to publish network resources using the LDAP protocol. The global graph structure is such a resource that is published. If an offer or reservation is made by a NetworkManager, the global graph structure is loaded from LDAP to get the most actual form of it. Any changes regarding this structure can be loaded back to LDAP. The graph structure (local and global) are implemented using the graph package.
The last package used in the NetworkManager is visibleGraph and realizes an administration interface for the user.

3.8.5 GUI

To simplify administrative work, a graphical user interface for the NetworkManager was implemented. This GUI should enable the administrator of the managed domain both to view and edit the structure of the domain. Additionally the reservations for all network links can be viewed.

All features of the GUI will be explained briefly subsequently.

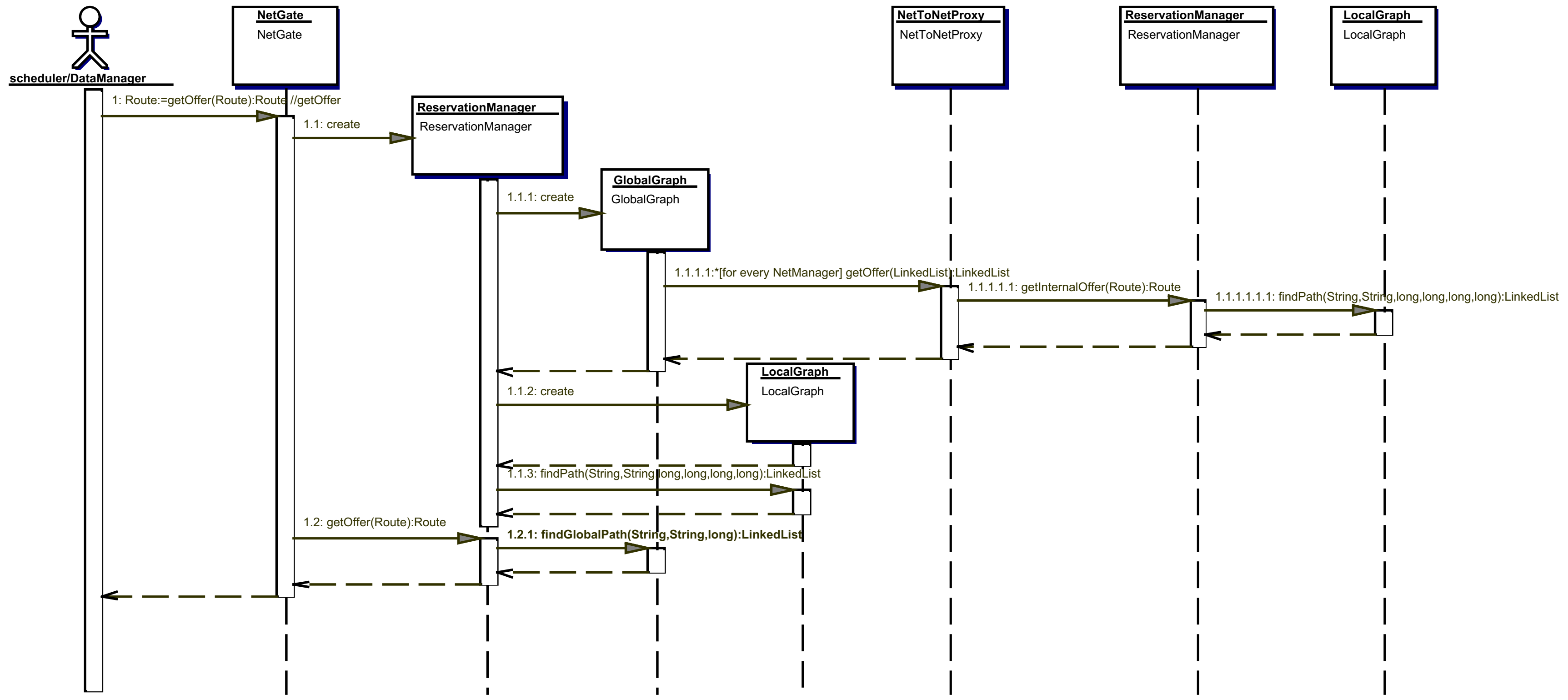

scheduler/DataManager
NetGate
NetGate
ReservationManager
ReservationManager
GlobalGraph
GlobalGraph
NetToNetProxy
NetToNetProxy
ReservationManager
ReservationManager
LocalGraph
LocalGraph
LocalGraph
LocalGraph
1: Route:=getOffer(Route):Route //getOffer
1.1: create
1.1.1: create
1.1.1.1:*[for every NetManager] getOffer(LinkedList):LinkedList
1.1.1.1.1: getInternalOffer(Route):Route
1.1.1.1.1.1: findPath(String,String,long,long,long,long):LinkedList
1.1.2: create
1.1.3: findPath(String,String,long,long,long,long):LinkedList
1.2: getOffer(Route):Route
1.2.1: findGlobalPath(String,String,long):LinkedList

Figure 3.53: Example getOffer().

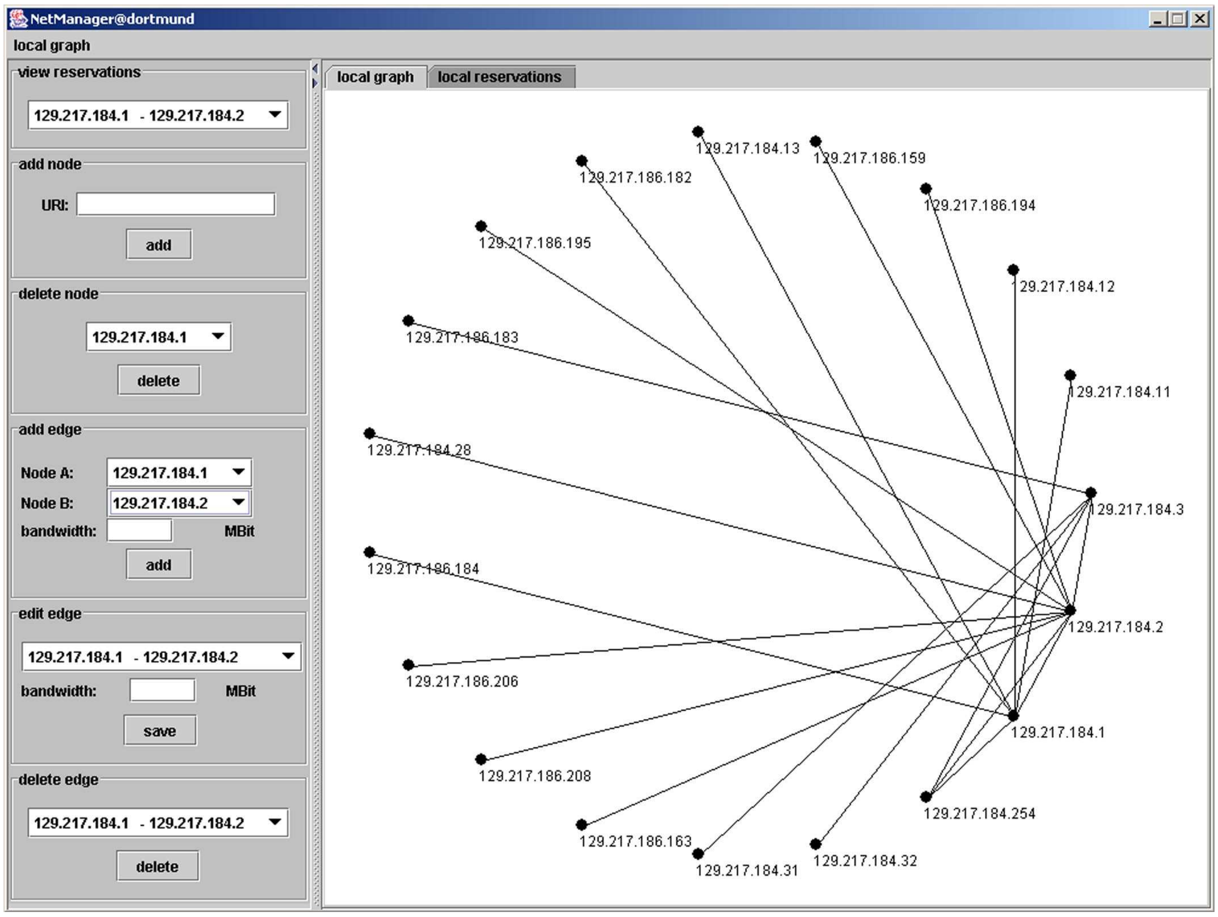


Figure 3.54: GUI of the NetworkManager.

3.8.5.1 Using the GUI

For a better understanding of the following explanations, please have a look at Figure 3.54. The GUI is divided into two parts by a split pane. The right part of the pane shows a graphical representation of the loaded local graph or the reservations of the network links. The user can choose between these possibilities by pressing the corresponding tab. The left part contains sub-panes which enable the user to edit the graph or choose a network link whose reservations should be shown in the right pane. Starting the GUI a graphical representation of the local graph specified in the properties file is presented.

The graph just displays all network nodes and the links between them. To add a new node to the graph the user just has to enter the URI of the new node and finish the action by pressing the button labeled ”add”, the node will appear right away in the graph. To delete a Node the correspondent node must be selected by means of the combo box inside the pane ”delete node”. After confirming the selection by pressing the button ”delete” the node and all adjunctive edges will be deleted.
To add an edge, the user just has to select the nodes the edge should connect and enter the bandwidth of the edge. After pressing the ”add” button, the changes will appear right away. Of course the entered bandwidth should be not greater than the actual bandwidth of the network link as it represents the reservable amount of bandwidth of the link. The user must take care of this herself as the GUI can not validate this.

Furthermore the edge must connect two different node, otherwise a warning would appear and the edge would not be created. To edit an edge respectively its bandwidth, the edge should be selected by means of the corresponding combo box inside the pane ”edit edge” and a new value for its bandwidth can be

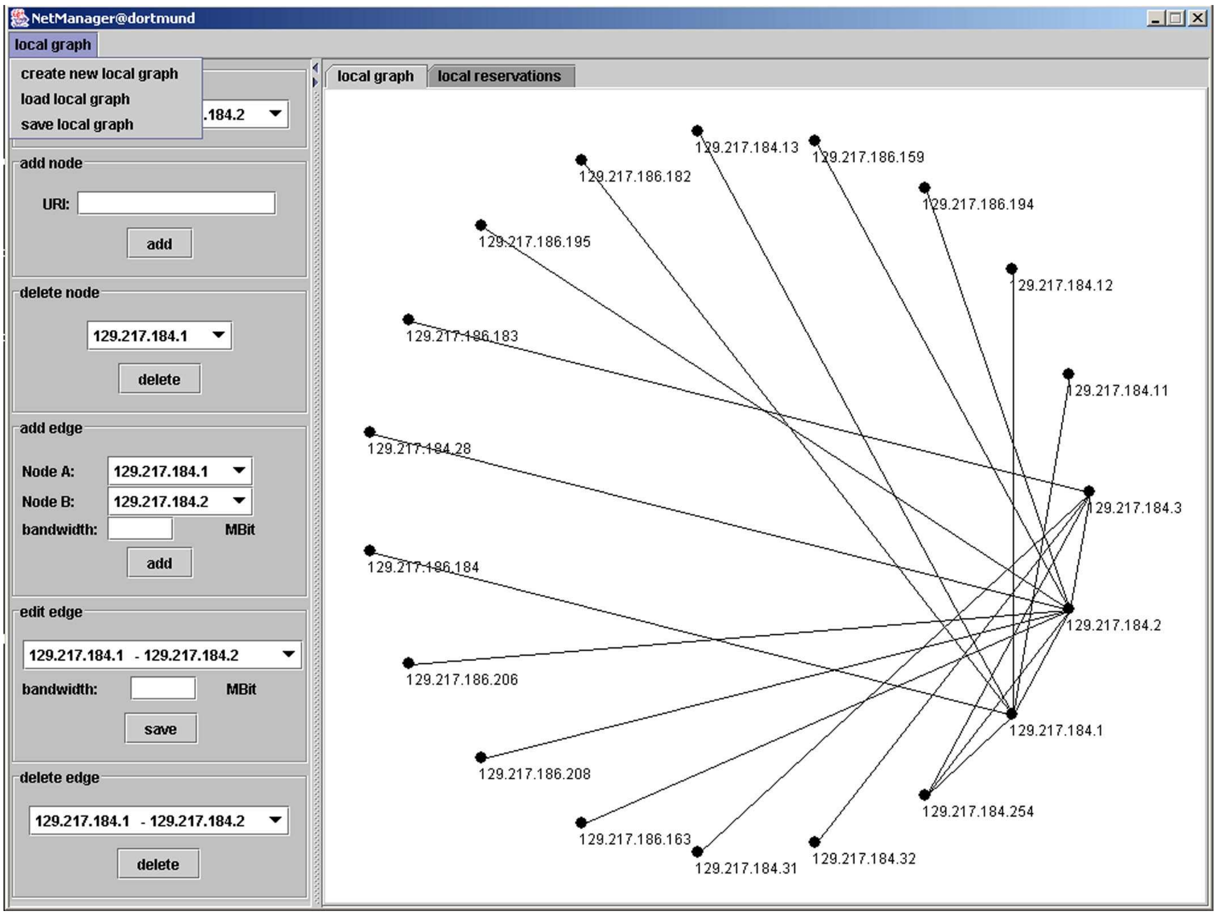


Figure 3.55: Menu of the GUI.

entered. To confirm the changes, the button labeled with "save" must be pressed.

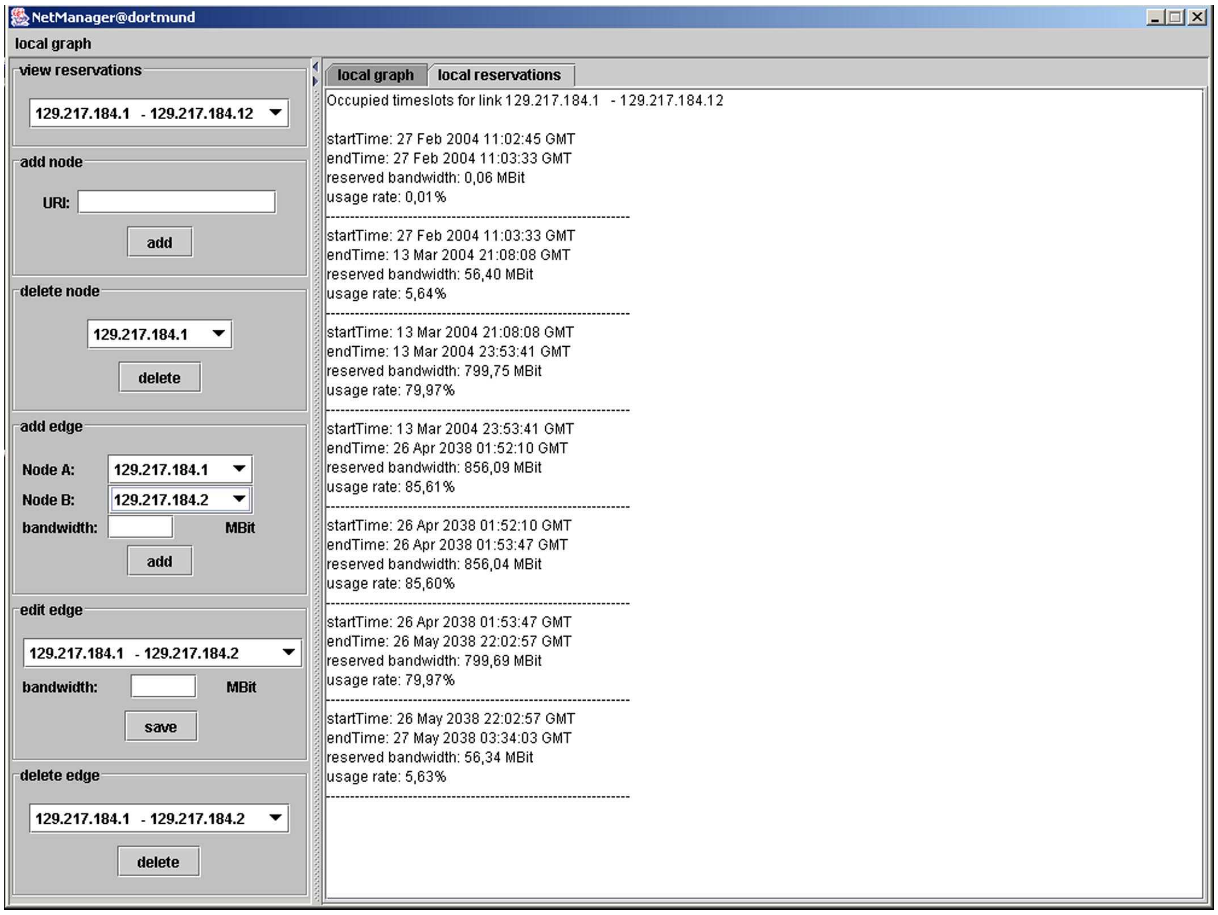
Again the user should not increase the bandwidth to an amount greater than the actual available bandwidth of the link. She can reduce the amount of bandwidth only if the new value does not affect reservations already made for this link. Otherwise a warning will appear and the changes will not be taken over.

To delete an edge, the edge just has to be selected by means of the combo box inside the panel labeled "delete edge". Of course this selection has to be confirmed.
All changes to the graph appear right away in the graphical representation and also affect the running system. But to make the changes persistent, the graph has to be saved.

To save the graph the user must select "save local graph" from the menu "local graph", see Figure 3.55. Then she can specify a location to store the graph in. It should be kept in mind that the saved graph will be only loaded at the next start of the NetworkManager if it is stored in the location specified in the configuration file or if the configuration file is changed. Using this menu the user can also create a new, empty local graph or load an another existing local graph.

A new graph has to be created only when constructing a new domain. Loading an existing graph is mainly needed for testing purpose.

As said before besides editing the local graph the user can also use the GUI to view the reservations for every network link of the managed domain. To do so she has to select "local reservations" in the right tabbed pane. Then an edge should be selected by means of the combo box inside the pane labeled "view



```
Occupied timeslots for link 129.217.184.1  - 129.217.184.12

startTime: 27 Feb 2004 11:02:45 GMT
endTime: 27 Feb 2004 11:03:33 GMT
reserved bandwidth: 0,06 MBit
usage rate: 0,01%
----------------------------------------------------------
startTime: 27 Feb 2004 11:03:33 GMT
endTime: 13 Mar 2004 21:08:08 GMT
reserved bandwidth: 56,40 MBit
usage rate: 5,64%
----------------------------------------------------------
startTime: 13 Mar 2004 21:08:08 GMT
endTime: 13 Mar 2004 23:53:41 GMT
reserved bandwidth: 799,75 MBit
usage rate: 79,97%
----------------------------------------------------------
startTime: 13 Mar 2004 23:53:41 GMT
endTime: 26 Apr 2038 01:52:10 GMT
reserved bandwidth: 856,09 MBit
usage rate: 85,61%
----------------------------------------------------------
startTime: 26 Apr 2038 01:52:10 GMT
endTime: 26 Apr 2038 01:53:47 GMT
reserved bandwidth: 856,04 MBit
usage rate: 85,60%
----------------------------------------------------------
startTime: 26 Apr 2038 01:53:47 GMT
endTime: 26 May 2038 22:02:57 GMT
reserved bandwidth: 799,69 MBit
usage rate: 79,97%
----------------------------------------------------------
startTime: 26 May 2038 22:02:57 GMT
endTime: 27 May 2038 03:34:03 GMT
reserved bandwidth: 56,34 MBit
usage rate: 5,63%
----------------------------------------------------------
```

Figure 3.56: Reservations of a specified network link.

reservations". Then all reservations for the selected edge can be seen in the right pane, see Figure 3.56.

3.8.6 Unsolved Problems

There are two problems that could not be solved because of the lack of time:

- we were not able to add the NetworkManager to GridSched's simulation on time
- as mentioned before, we were also not able to realize a forecast service, because of the problems connected to it

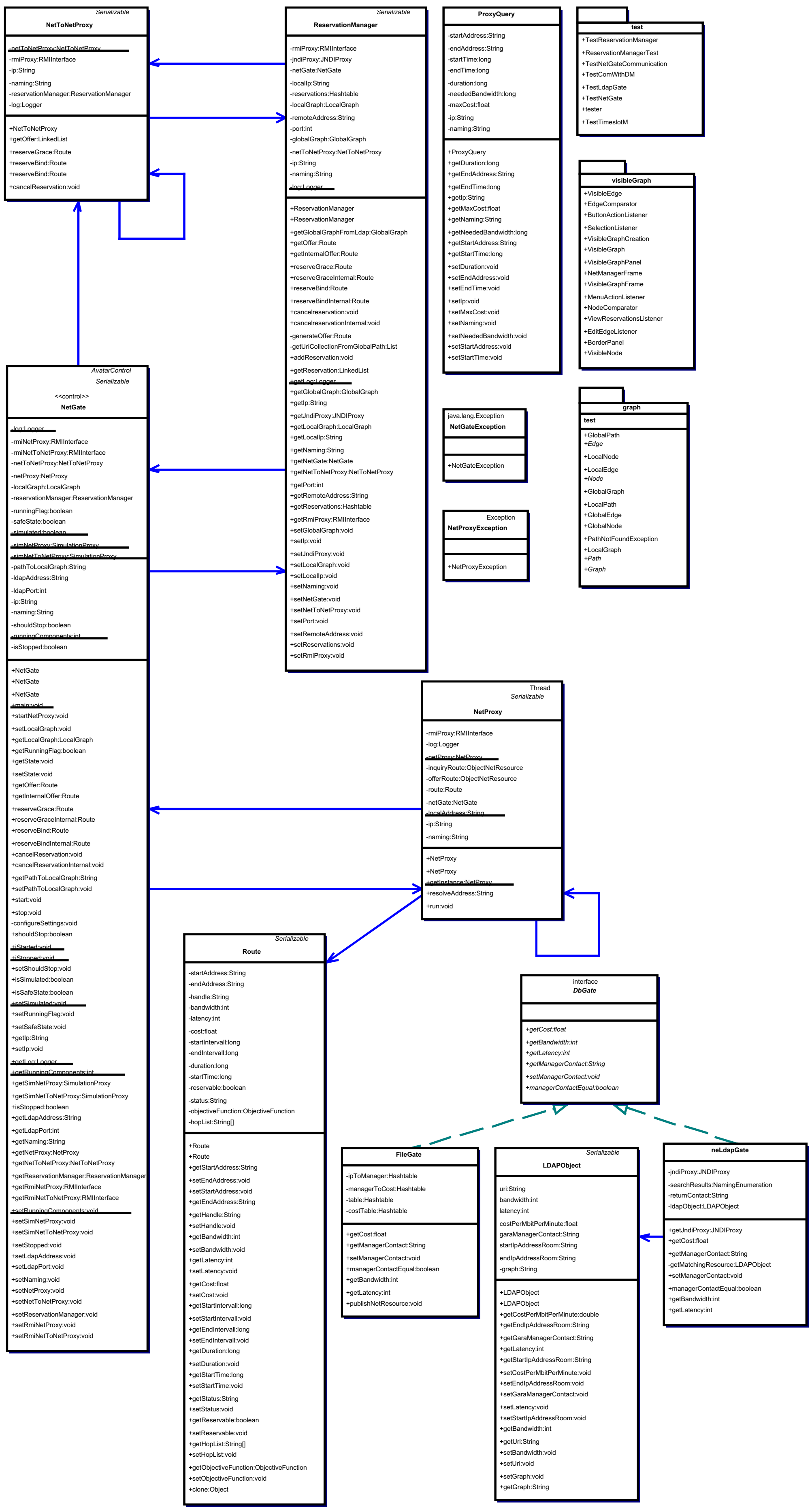


Figure 3.57: Class Diagram NetworkManager.

3.9 Policy

In a distributed, heterogeneous environment like a Computational Grid, powerful means for user authentication and authorization are essential. Participants have to protect their systems from unauthorized access and want to remain in control of the resources they offer to the Grid. The software must provide appropriate functionality which has to be easy, understandable and robust . A complex system is likely to raise doubt and concern about whether the configuration was really done correctly.
A software that fails to meet the requirements mentioned above will be rejected by the market.

3.9.1 Requirements

In practice, different users request access to resources.
The owner of the resources now deploys a policy that contains the following elements:
Users that identify certain users, **groups** that help to state rules for more than one user simultaneously, and **resources** that reflect the hard- and software the owner offers on the Grid. Apart from that, there are **permissions**, that state what is allowed. The rules that define the permissions for a user and a certain resource are called **policy**.

3.9.2 Idea

It is necessary to be able to use wild cards and give rules a well defined order of precedence to meet the requirements defined above. This is achieved by implementing the policy as a tree. Nodes are labelled with (user, group, resource) and represent a permission.

When handed (user, group, resource) and being questioned for the corresponding permission, the tree is traversed until a node has no successors that also fit the (user, group, resource). From the root of the tree towards the leafs the labels of the nodes become more specific. The root of the tree therefore is labelled (See Figure 3.58), thus matching any user in any group for any resource.

3.9.3 Implementation

The implementation mainly uses two classes: `Policy` and `PolicyManager`. The class `PolicyManager` uses an inner class called "entry" which wraps `Policy`-objects into tree nodes, thus enabling the `PolicyManager`-class to build a tree as described above.
`PolicyManager` returns the `Policy`(the applying rule) for a supplied (user, group, resource). This `Policy` can be questioned for the permission.
The reason for the `PolicyManager` not to return the permission directly is extensibility. The class `Policy` is likely to be augmented by new functions, providing more functionality to the user. The `PolicyManager`-class is completely independent from them.

3.10 Deployment of GridSched's Components

In this section the allocation of GridSched's components respectively its services to machines within the managed Grid environment is described fundamentally. Additionally an overview of all GridSched's communication interfaces and protocols is given. The GridSched's concept regarding Grid domains is explained as well.

A Grid domain is a domain in a Grid environment. It represents a bounded network address space. When GridSched is running in a Grid domain it is called a managed Grid domain. A member of a domain is capable to address all other members within the same Grid domain. A domain can contain sub-domains to organize members in logical or spatial sections.

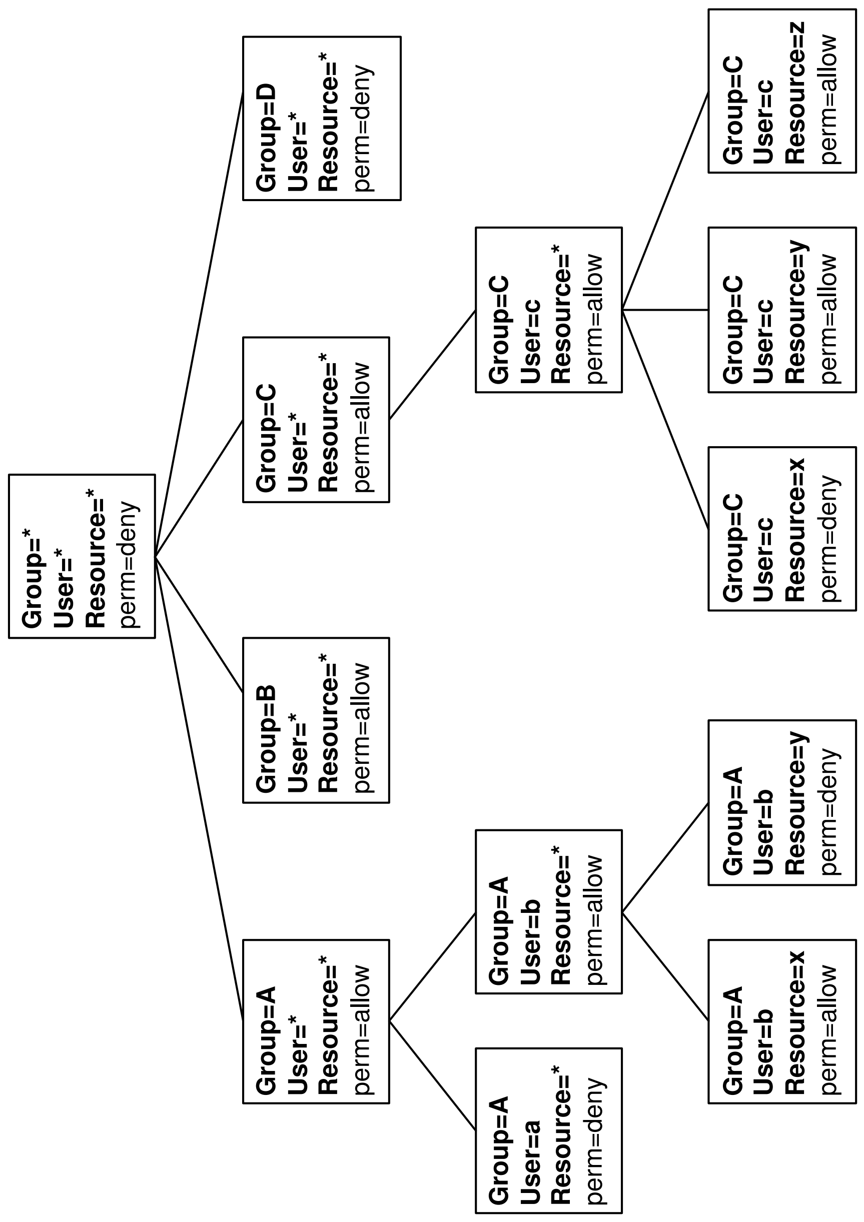

Group=* User=* Resource=* perm=deny
Group=D User=* Resource=* perm=deny
Group=C User=* Resource=* perm=allow
Group=B User=* Resource=* perm=allow
Group=A User=* Resource=* perm=allow
Group=C User=c Resource=* perm=allow
Group=C User=c Resource=z perm=allow
Group=C User=c Resource=y perm=allow
Group=C User=c Resource=x perm=deny
Group=A User=b Resource=* perm=allow
Group=A User=a Resource=* perm=deny
Group=A User=b Resource=y perm=deny
Group=A User=b Resource=x perm=allow

Figure 3.58: The tree-structure of the policy.

It is supposed that a carrier running GridSched is a resource provider and an owner or carrier of a Grid domain. The range of GridSched's control is fully decided by its carrier. She is free to install a Local Scheduler on certain or all domain's machines to offer their resources within the managed domain.

In GridSched components are differentiated by existence as one or many instances. In a Grid domain managed by GridSched there exists only one instance of the directory service, PolicyManager, Communication component and the ClientGUI. Bob, Titan, DataManager, NetworkManager and ServerGUI are all GridSched's services with more than one instance existing in a managed Grid domain.

An instance of Bob, GridSched's Local Scheduler represents all other Bob's running in a single subdomain of the managed Grid domain.

A single instance of GridSched's Super Scheduler is able to be addressed by members of more than one managed sub-domain. Moreover there are several instances allowed to exist within a single managed domain. So it is possible that a single Bob may be addressed by more than one Super Scheduler.

Different kinds of GridSched's services and components are able to run on a single machine in any combination. On the other hand each existing GridSched service may be deployed on its own machine.

To describe the communication between GridSched's components different protocols and methods may be distinguished.

Most of GridSched's components use the communication method provided by GridSched's communication component using a specific data structure mentioned before and RMI communication methods. The communication links between the involved components are lined up as pairs representing all possible combinations of existing component instances within a managed Grid:

- Local Scheduler (Bob) and Super Scheduler (Titan),
- DataManager and NetworkManager,
- Local Scheduler (Bob) and DataManager.

Some GridSched services use unique RMI-based communication models because other kinds of data structures are needed to be transferred. This is related to the following combinations of GridSched components:

- ClientGUI and Titan,
- ServerGUI and Bob,
- ServerGUI and Titan,
- ServerGUI and DataManager,
- ServerGUI and NetworkManager.

A third communication method used in GridSched is represented by the LDAP protocol. It is used by Bob and Titan to access the directory service running in a managed Grid.

3.11 Summary of GridSched's Architectural Features

GridSched is designed to be a middleware solution bringing together the providers and users of Grid resources and providing Grid Scheduling functionality.

All of the system's parts are designed as unique components with specific tasks. Combined with well-defined interfaces it is possible to set up a running GridSched system using different combinations of it's components. Moreover it is possible to swap GridSched's components by other component implementations to fit GridSched to special needs of it's carriers.

The concept of distributed scheduling is realized by GridSched's DataManager, NetworkManager, Local Scheduler (Bob) and Super Scheduler (Titan). Bob and Titan exchange reservation data by using GridSched's communication component. Titan is designed to use exchangeable scheduling strategies using different heuristic scheduling algorithms implemented in Titan's sub-components.

All information on resources available to the scheduling components are stored in a centralized database called directory service. Its access is realized by LDAP.

GridSched's DataManager uses a SQL-based database to manage data files available and used in a managed Grid domain and their properties. DataManager provides interfaces to Titan and GridSched's NetworkManager to enable scheduling of data resources.

Like Bob GridSched's NetworkManager works above local hardware schedulers which are realizing reservations in Grid's hardware layer. It is controlled by DataManager via the communication interface provided by GridSched's communication component. NetworkManager integrates two different data structures to represent the existing network infrastructure in a managed Grid environment: in the global graph its nodes represent domains while Local graph's nodes represent machines. In both graphs the edges represent network connections. All nodes and edges store network resource and reservation information.

The communication component of GridSched provides an universal communication model used by most of the components. It is described by a client-server structure. Both communication partners are able to act as a server and a client. Commands and it's Replies are transferred between the components involved in the scheduling and reservation process. These communication entities are encapsulated in a data structure called MessageEnvelope. All commands include parameters which are part of the job description.

GridSched's job description is a descriptive language which is used globally by most of the components to process jobs and to schedule needed resources. It describes the needed resources and the costs caused by resource reservations. It's hierarchical structure makes it possible to compose a job from sub-jobs.

A user interface to the Super Scheduler is provided by the ClientGUI. It utilizes the job description language to define and manage jobs by it's users.

In ServerGUI a hierarchical data structure is used to process the XML-based configuration files. Control classes capture the read-write-access to the configuration files and to start and stop GridSched's executable components called GridSched services. Two classes implement graphical user interfaces built up dynamically. It's control elements trigger the control classes methods.

The PolicyManager implements authorized access to GridSched's functionality and the managed resources. Policies describe the authorization as formally defined rules combining users, user groups, resources and permissions. Policies are represented by a tree data structure which is used to store permission values for certain user and resource combinations.

Chapter 4

Simulation in GridSched

4.1 Concept of GridSched's Simulator

This section deals with the simulation component of the GridSched project. Firstly a short explanation about why it is necessary to simulate is given. After that, different kinds of simulation techniques are described and it is explained which simulation approach was chosen for GridSched. Then the architecture of the simulation component and the integration of the other components of GridSched into the simulation are described.

After designing and implementing a system there is a need to prove that the system produces correct and good results. Scheduling tries to optimize. Therefore it is interesting to evaluate the quality of the included scheduler in contrast to others. That means to find out how good the results are that the GridSched system produces compared to other scheduling systems.

There are several approaches to evaluate a system which could be divided in two main directions.

- **Theoretical analysis of systems and algorithms** means to determine the performance of algorithms or systems by mathematical methods. The goal of these examinations is to prove lower or upper bounds. These methods are difficult in combination with scheduling algorithms because they are very complex and difficult to handle. On top of this it is often just possible to prove worst-case performance, while it is often more interesting to know about the average performance of a scheduling algorithm.
- **Practical analysis of systems and algorithms** includes many different methods to evaluate a system or an algorithm, for example benchmarking of the real system and testbeds. One of the most interesting practical methods is simulation. The goal of simulating a system is to describe, explain and to predict the behavior of the real system. One of the advantages of a simulation is that it can be used in cases where other analyzing methods fail because the examined system does not yet exist or the examined course of events is either too slow or too fast to be recognized in reality. A simulation is often a cheaper, less dangerous, more flexible and faster alternative and in some cases even the only possible method to evaluate a system. A model of the system must be programmed to run a simulation.

There are several different simulation techniques which can be divided in

- **Continuous simulation** which uses mathematical models, mainly in form of differential equations. In this kind of simulation the state of the system could be computed for every moment.
- **Discrete simulation** where time is proceeding in discrete steps. That means the state of the system can only be determined for certain times. The class of discrete simulation can be further divided into event-based and time-based simulation. Time-based simulation means that the simulation time is always proceeding in determined time intervals while event-based simulation means that the simulation time is proceeding because of the occurrence of events.

A discrete event-based simulation has been chosen to evaluate the GridSched project to meet all requirements regarding the GridSched project. This kind of simulation technique offers the possibility to jump from one event to another when nothing happens between two events which is relevant for the system. That means that you are able to use workloads, for example of a whole year, which can be processed in a much shorter time because the time a simulation needs to run depends on the amount of events being processed and not on the period of time that is covered by the used workload. Time between events is not interesting for the system because there is nothing happening which could affect the state of the system. A scheduling system is such a system in which changes of the state occur when a new job is submitted or a job is executed and can leave the system. These are not the only occurring events. While processing a job new events can appear which are relevant for the system, like messages between parts of the simulation model.

A Grid in general consists of host computers, where services are running to provide its familiar functionalities. These are connected with network connections to exchange information. To mirror them, the simulation maps host computers to **Nodes** and services to **Avatars** resulting in a **virtual** Grid. These are controlled by the simulation via Java RMI mechanisms. The design is applied to perform a simulation scene on a single workstation. According to the fact that services operate externally and each form is running exclusively on a computer under real conditions, the Nodes representing a Grid scene are multiplexed to a single machine by a timesharing mechanism similar to applied operation system concepts. Thus one `Node` including all associated Avatars is simulated at once. Only Titan as a part of simulator is excluded from this concept, it is even running exclusively in the simulation program. That means, that only one instance of a super-scheduler may exist in a simulation turn.

4.2 Design and Implementation

The GridSched simulation consists of four parts, the initialization and control component, an integrated database called 'Okeanos', the processor 'Elysion' and an evaluation facility. All components are kept together by the `Simulation` class (see Figure 4.1), containing all references of control classes.

To obtain a closer relationship to reality conditions of a Grid, the design approach is to change as least as possible to the GridSched services. Thus services as Bob and DataManager are supposed to run separately from the simulation program in their familiar surroundings, as independent computer programs running on the underlying operation system. But to take control over them, they have to be slightly modified. This is done by implementation of the `SimulationProxy` class, assuming several functionalities based on communication, persistence, and initialization of GridSched services. Furthermore modifications to Bob and DataManager became evident, i.e. running jobs virtually. A service controlled by the simulation environment is mentioned as an **Avatar**. The sections below describe the implementations of all components of the simulation.

4.2.1 Initialization and Control

There are some Java classes which are responsible for initializing and controlling the simulation (see Figure 4.1). These are

- `SimulationControl` This class is responsible for controlling the simulation. It initializes all the needed components to run a simulation. It also contains methods to start, stop, resume, create, save and load a simulation. If such a method is called, the `SimulatorControl` directs the call to the responsible class like `Init`, `Elysion` or `Persistence`.
- `Init` Builds up the simulation environment. There are two possibilities to build up a simulation. One is to enter all needed components via the Simulation GUI (for more details see Section 4.5). Then the corresponding Java objects are created and the simulation environment will be set up. An other opportunity is to use an XML file in which the simulation environment is described. Then the XML document is parsed and the needed Java Objects are created. Therefore the `Init` class has methods like `createNodes()`, `createLinks()`, `createBobResources()` and a lot more to produce the Java objects (see also Section 6.6.1). The produced nodes for example are stored in a database

called Okeanos (see also Figure 4.2). Its functions are saving and managing of nodes. It covers all functionalities concerning the management of nodes, i.e. the registration and removal of nodes in the simulation scene, or modifications on their attributes. Okeanos uses a linked list to store the nodes.

- `Persistence` Saves and loads the state of a simulation. Therefore two methods are existing:
  - The `save()` method, stores the actual state of the simulation which is done by putting the state of the `SimulatorControl` in an `ObjectOutputStream` which is then saved into a zip file which contains not only the actual state of the simulation but also Avatar state files, so that Avatars can resume their work in the state they had when they were stopped, a log file and if existing the XML file by which the simulation was build.
  - The `load()` method, reads the content of the zip file and unzips the files. Then the state of the simulation is loaded by producing an `ObjectInputStream`. Then the states of the saved simulation components are reestablished.

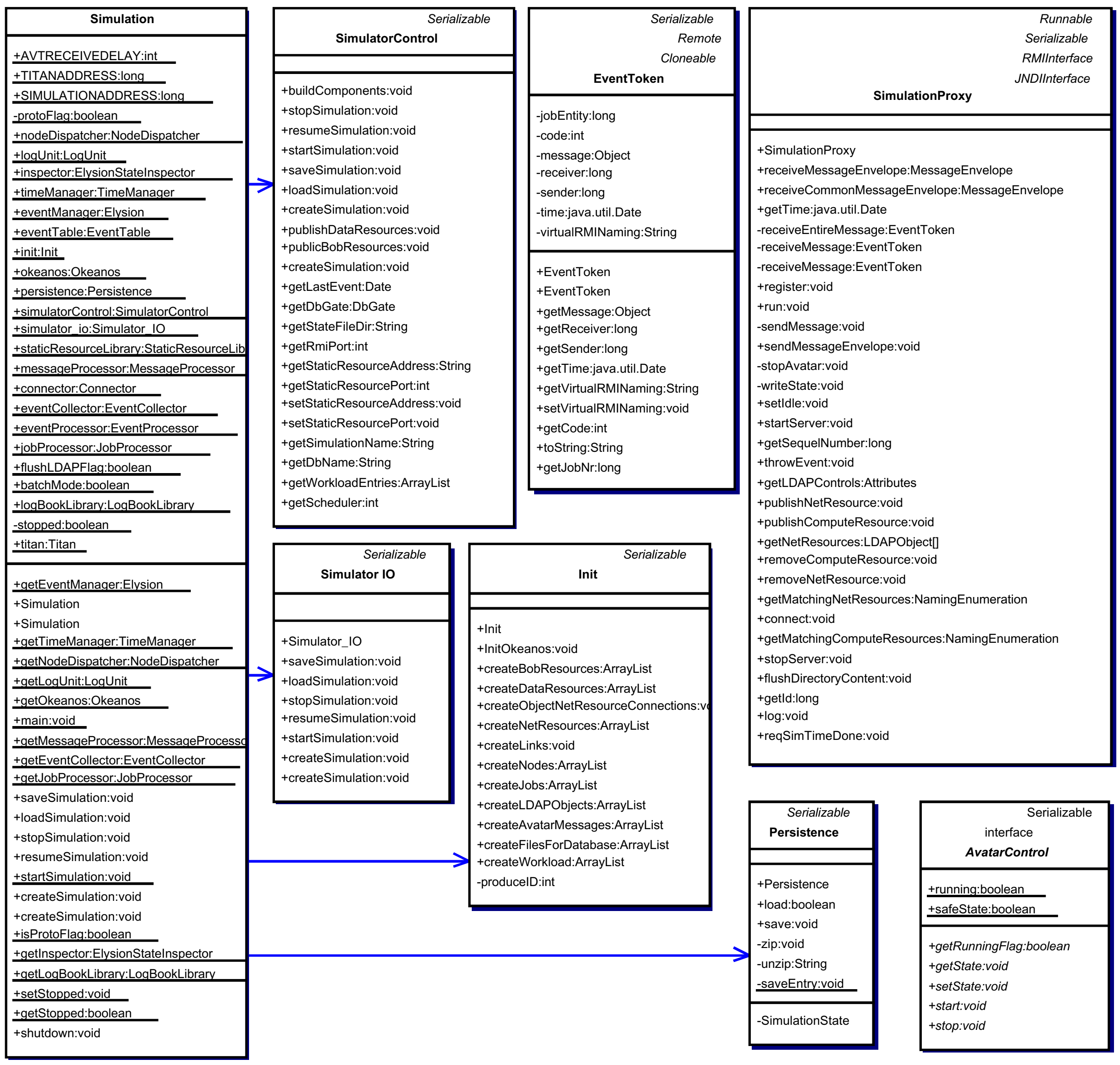


Figure 4.1: Simulation main package classes.

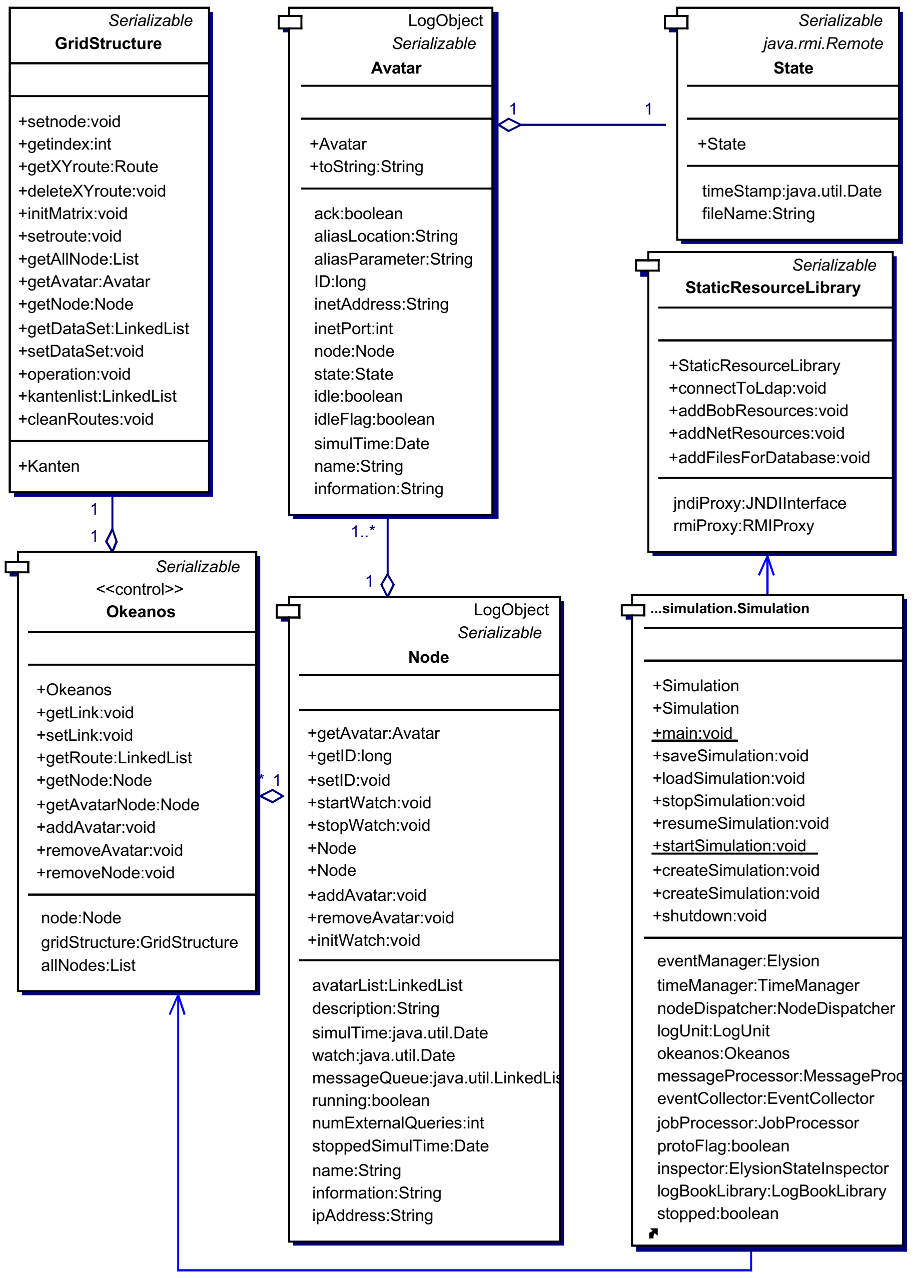


Figure 4.2: Okeanos classes.

4.2.2 Processor 'Elysion'

The title **Elysion** is given by the Greek mythology, a land in another dimension, connected to Earth in a way that if Elysion is destroyed, then so is Earth. The analogy to a simulation environment is self-evident. Several major assignments are complied by Elysion, setting up virtual computers (mentioned as **Nodes**) and a network environment for Avatars, to log their activities and process simulation incidents. As a design approach, Elysion is partially event-controlled, resulting in a less complex object-model. Its role in the simulation is to process jobs, as they would run in reality. Furthermore it provides mechanisms to evaluate a simulation turn.

Elysion is parted into several Java implemented classes (see Figure 4.3), structured by its given assignments:

1. `JobProcessor` Manages the processing of a single job, and contains all jobs being committed as a working-set.
2. `JobEntity` A data-structure representing a single job and mirrors its state.
3. `MessageProcessor` Mediates the message-traffic between Avatars and Elysion, and encapsulates the GridSched communication facility.
4. `EventProcessor` As it was mentioned, that Elysion is partially event-driven, it processes thrown events, and engineers their effects.
5. `EventCollector` Accepts events from any components to collect them. It notifies the EventProcessor to take and process them.
6. `JobFactory` A special class, providing a static method to generate up to thousands of jobs by using templates, which can defined in the simulation´s input.
7. `NodeDispatcher` The core unit to control booting and shutdown of Nodes (see Section 4.3.0.6).
8. `Elysion` The Interface of the processor enabling control over simulation progression.
9. `Connector` The communication agent to manage information flow between the processor and Avatar´s associated to Nodes, which are set up by Elysion.
10. `TimeManager` An entity containing simulation time, and to measure its time advance.
11. `Titan package` Titan is a part of Elysion in this context, by providing its scheduler and job-processors. According to the simulation approach, the scheduler is the entity to be evaluated.

4.2.2.1 Virtual Networking

Working on a virtual Grid requires the appliance of network in a virtual way. The first modification is the naming of network addresses, the simulation uses integer values for addressing network entities instead of IP numbers, these are identical to the identifier of Nodes and Avatars (see Figure 4.6). Routing is performed by the `MessageProcessor` class, the communication unit of the processor. It manages message queues of all Nodes, where all associated Avatars can send or receive messages. Furthermore an RMI interface is defined to access all incorporated operations that are performed via network. From Avatar´s point of view, it is accessed by the `SimulationProxy` (see Figure 4.1) hosted in each service running in the simulation environment. On the one hand, network activities, affecting only the current Node are non-critical since all participants are running and can react immediately. On the other hand, messages leaving the current Node must be handled especially, because the receiver is not active and cannot react. So the message is stored into the message queue of the addressed Node, which is scheduled to start, and the requesting Node will be shut down before. This context switch of Nodes is required to process this event completely. This functionality is hosted in the `NodeDispatcher class`, and is parted into two phases:

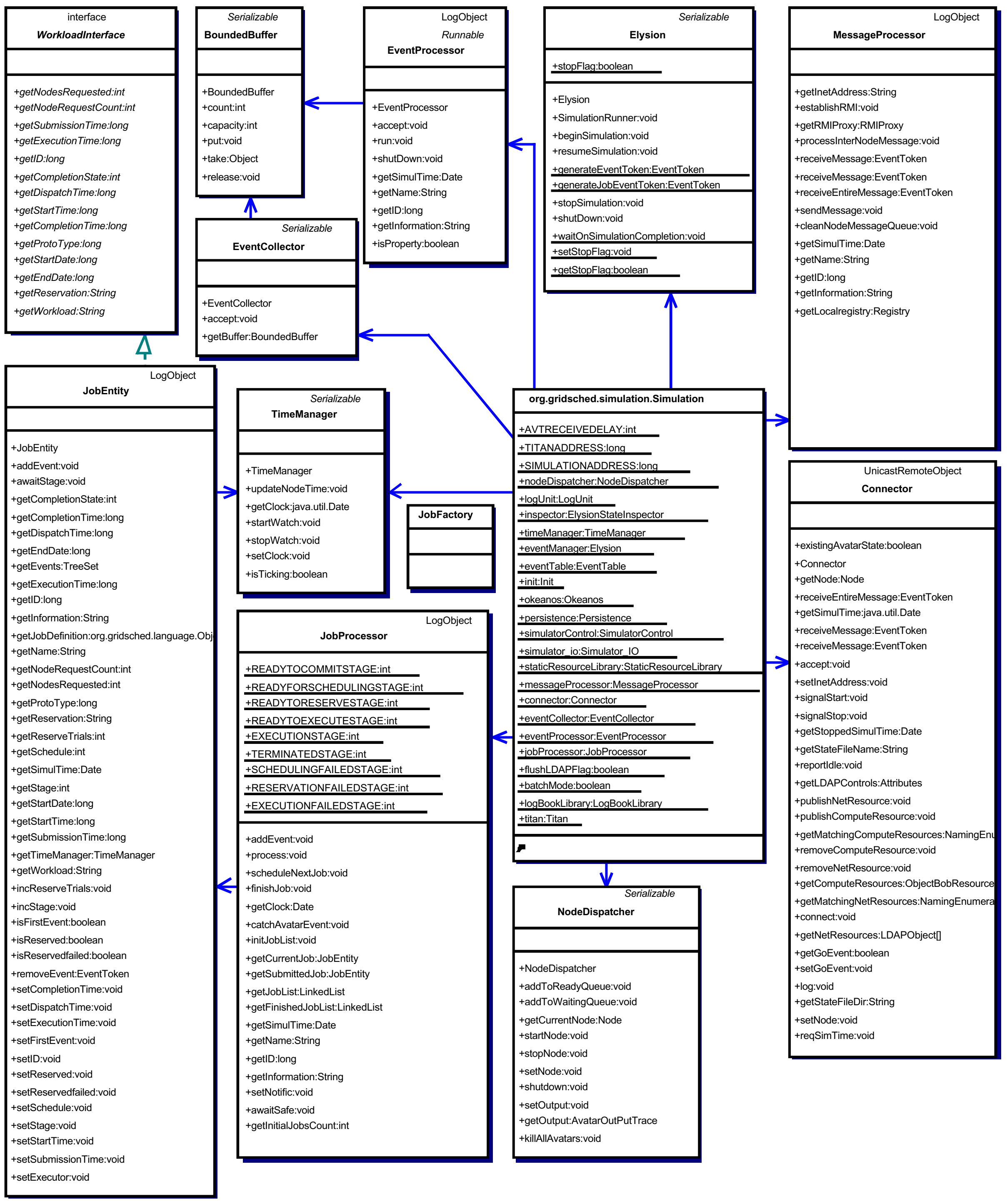


Figure 4.3: Elysion Classes.

- **Starting a Node** It is invoked by calling the `startNode()` method (see Figure 4.4). At first its message queue is restored from its associated `Node` object. All program files of the associated Avatars will be executed by the operation system in a special simulation mode, these are idle at this moment and are waiting for simulation controls. The `Connector` object becomes active and sends '`clear to run`' messages (represented by `EventToken` class) to all Avatar processes and waits for their acknowledgments (see Figure 4.4). After that, the Nodes simulation time is synchronized to the global simulation time and its watch is started. Finally a 'go-event' flag is set, signaling the Avatars to start their operation.

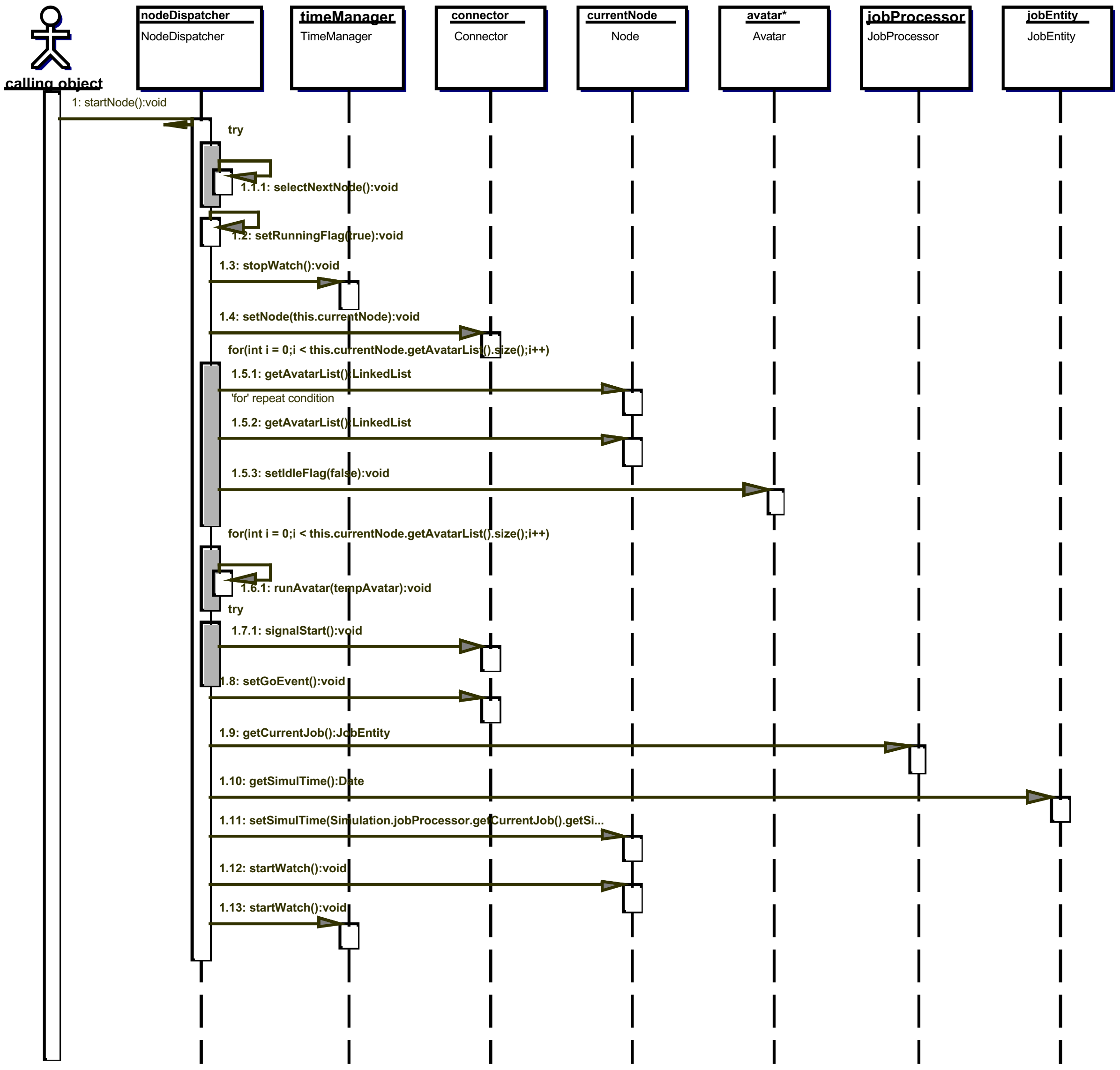

calling object
nodeDispatcher
NodeDispatcher
timeManager
TimeManager
connector
Connector
currentNode
Node
avatar*
Avatar
jobProcessor
JobProcessor
jobEntity
JobEntity
1: startNode():void
try
1.1.1: selectNextNode():void
1.2: setRunningFlag(true):void
1.3: stopWatch():void
1.4: setNode(this.currentNode):void
for(int i = 0;i < this.currentNode.getAvatarList().size();i++)
1.5.1: getAvatarList():LinkedList
'for' repeat condition
1.5.2: getAvatarList():LinkedList
1.5.3: setIdleFlag(false):void
for(int i = 0;i < this.currentNode.getAvatarList().size();i++)
1.6.1: runAvatar(tempAvatar):void
try
1.7.1: signalStart():void
1.8: setGoEvent():void
1.9: getCurrentJob():JobEntity
1.10: getSimulTime():Date
1.11: setSimulTime(Simulation.jobProcessor.getCurrentJob().getSi...
1.12: startWatch():void
1.13: startWatch():void

Figure 4.4: `startNode()` sequence diagram.

- **Stopping a Node** Invoked by calling the `stopNode()` method (see Figure 4.5). It calls the `Connector` to stop the actual local simulation time and emits 'stop signals' to all running Avatars. The acknowledge messages contain the file descriptors for their state files (refer to Section 4.3), by receiving them all Avatars switch into idle state and could be interrupted . Then the LST (local simulation time) is saved by the `NodeDispatcher` and all Avatar processes are terminated (see Figure 4.5). At least the message queues of the `Connector` are saved into the corresponding `Node` objects, and the `Connector` object is disposed.

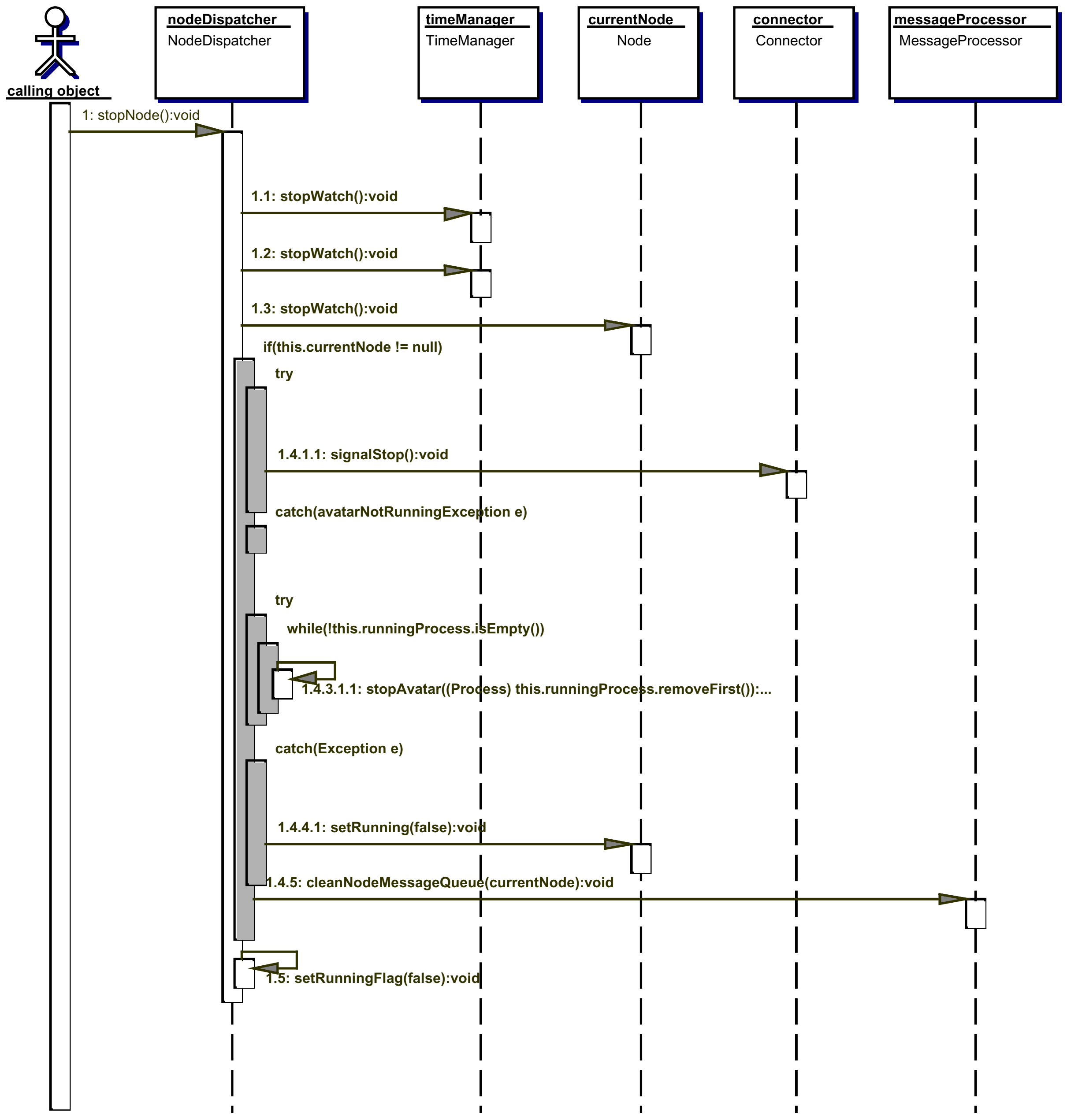

calling object
nodeDispatcher
NodeDispatcher
timeManager
TimeManager
currentNode
Node
connector
Connector
messageProcessor
MessageProcessor
1: stopNode():void
1.1: stopWatch():void
1.2: stopWatch():void
1.3: stopWatch():void
if(this.currentNode != null)
try
1.4.1.1: signalStop():void
catch(avatarNotRunningException e)
try
while(!this.runningProcess.isEmpty())
1.4.3.1.1: stopAvatar((Process) this.runningProcess.removeFirst()):...
catch(Exception e)
1.4.4.1: setRunning(false):void
1.4.5: cleanNodeMessageQueue(currentNode):void
1.5: setRunningFlag(false):void

Figure 4.5: `stopNode()` sequence diagram.

The sequence depends on the state of the `NodeDispatcher`, is no Node running, the context switch is performed without stopping a Node. Although Titan is a permanent instance, its attitude is similar to a Node, when a message is sent to them, the current `Node` is also interrupted due to the expectation that the next `Node` that will be simulated will change. All virtual network activities are logged by the `MessageProcessor` (see Figure 4.3) and can be watched at the provided graphical user-interface (see Section 4.5).

4.2.2.2 Job Processing

The simulation object is evidently the procession of jobs as they would occur in reality. To simulate them, each job´s life-cycle in the simulation is parted into stages (see Figure 4.8), according to the ordering of incidents in the reality, when a job is committed to Titan. Finally, when a job has been simulated

Identifier	Entity
99	Titan
-2	Simulation Processor
-1	Log Unit
greater 0 divisible by 10	Node
(greater 0 divisible by 10) + 1	Bob
(greater 0 divisible by 10) + 2	DataManager
(greater 0 divisible by 10) + 3	NetworkManager

Figure 4.6: Virtual network addresses.

completely, its stage results in a final stage. The definition of stages are given in the *JobProcessor* class as final integer values, to provide a better comprehension, they are associated with names. So programmers can add or modify definitions with ease, but must consider semantics and algorithms of the `JobProcessor` and `EventProcessor` classes to keep functionality. An automate of the possible stage-transitions can be seen in Figure 4.7. An explanation of each stage is given below.

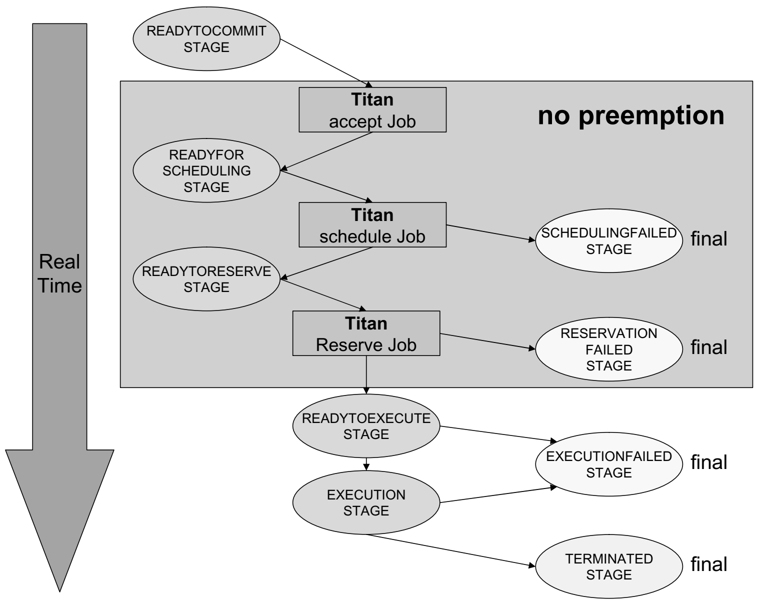


Figure 4.7: Job stage transitions.

As mentioned each job is represented by a **JobEntity** object, containing among others the job definition, analogue to real-time conditions, and a *submission time*, determining at what simulation time a job is to be simulated. So the hole working-set of jobs is the entire simulation program for a scene, it defines the sequence.
The *JobProcessor* takes control over the stages of every job in the working-set by programming among others `Titan` directly (see Figure 4.7). It uses its common interface for job scheduling and reservations of resources. First, the JobProcessor schedules a Job initially from its set (READYTOCOMMITSTAGE), then it will be committed to Titan. Afterwards (READYTOSCHEDULESTAGE) by invoking Titan´s *UserProxy* class´method `schedule()` and all job controls is passed to Titan temporarily. During its scheduling process, it will talk to several Nodes according to its policies. Each Node appealed will be started by the **NodeDispatcher**, and then shut down, when the Node has sent a response message. While this, the `JobProcessor` is sleeping. When Titan finishes a scheduling-process it will throw an Event to notify it, which evaluates the event, and determines the stage advance of the job (READY-TORESEVESTAGE), depending on the success of a schedule. If succeeded, the `JobProcessor` will invoke Titan to reserve. This process is similar to the scheduling mechanism, but reserved CPU Resources will

Process stages
READYTOCOMMITSTAGE
READYFORSCHEDULINGSTAGE
READYFORRESERVATIONSTAGE
READYTOEXECUTESTAGE
EXECUTIONSTAGE
TERMINATEDSTAGE
Final stages
SCHEDULINGFAILEDSTAGE
RESERVATIONFAILEDSTAGE
EXECUTIONFAILEDSTAGE

Figure 4.8: A short overview of all job stages.

request Elysion for simulation time, to process the execution of a reservation virtually. Afterwards, if all reservations are succeeded (READYTOEXECUTIONSTAGE), requests for simulation-time of resources are processed (EXECUTIONSTAGE) by the `JobProcessor`. When succeeded the job´s simulation is finished (TERMINATEDSTAGE). An explanation of each stage is given below, which includes other final stages, whose are reached, if something in the simulation process failed. A sequence diagram illustrates the implementation of job processing in the `JobProcessor` class (see Figure 4.9).

1. `READYTOCOMMITSTAGE` The initial Job stage, meaning each job in the working-set can be simulated.
2. `READYFORSCHEDULINGSTAGE` After a job has been committed to Titan, this stage is set.
3. `READYFORRESERVATIONSTAGE` Titan has thrown an event, that a schedule has been computed, and can be reserved.
4. `READYTOEXECUTESTAGE` All resources of a schedule have been reserved successfully.
5. `EXECUTIONSTAGE` The reserved *Timeslots* are being processed by the reserved CPU resource(s).
6. `TERMINATEDSTAGE` All reserved *Timeslots* has been processed and the simulation of this job has ended.
7. `SCHEDULINGFAILEDSTAGE` Titan has not found any suitable schedule, the simulation of this job is finally terminated.
8. `RESERVATIONFAILEDSTAGE` Titan could not reserve all scheduled resources, the simulation of this job is finally terminated.
9. `EXECUTIONFAILEDSTAGE` Any request computation for simulation-time by one or more CPU resources failed, the simulation of this job is terminated finally.

Elysion has the ability to simulate more than one job in single scene. So before a job can be simulated, the JobProcessor schedules a job for next. The inherent policy is quite simple, the job with the least time is selected, for more information about time synchronization, refer to Paragraph 4.2.2.3. When a job has been selected to be processed, the method `process()` in the JobProcessor class, takes control by calling further internal methods depending on the actual stage of it (see Figure 4.9). Due to the circumstance, that the simulation time of execution of reservation may overlap, some considerations in the design have been taken into account, to handle this. To prevent complicated implementations of *role-backs* in the simulation, preemption of jobs is performed when the simulation time windows of jobs overlap. But preemption is only supported when the actual job is in `EXECUTION or READYTOCOMMIT` stage (see Figure 4.7). Thus, jobs must not overlap others. This circumstance can be avoided, if the simulation input is so defined, that that case will never appear. Due to the fact, that the process of scheduling and reserving a job takes just tickles of time, and the execution of jobs is much longer, the submission times of jobs should have a distance of some minutes. In a simulation term of several days or month of simulation time, this matter is obviously negligible and carries no weight.

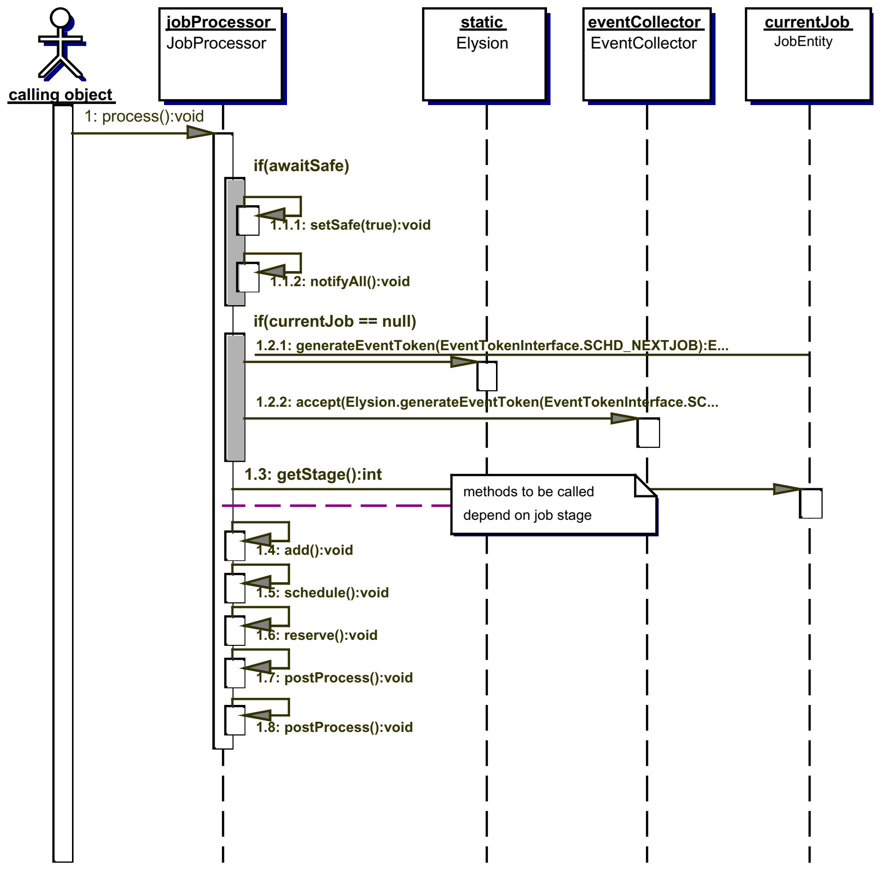


Figure 4.9: Job processing.

4.2.2.3 Time Synchronization

There are two different time-lines , the local simulation time (LST) and the global simulation time (GST). Each Job has a unique LST, whereas the GST is synchronized to the actual job that is processed. Additionally there is a watch for each simulation time entity to measure advance during simulation phases. The technical implementation has been realized in the `TimeManager` class and each instance is controlled by control class instances, such as the `JobProcessor` and the `MessageProcessor`. Several incidents make time synchronization necessary, three major activities can be performed then: a simulation time can be paused, resumed or set to a new time. The enumeration below lists some use-case examples.

1. **Job scheduling** After the `JobProcessor` has a new Job to be simulated selected, the current Job is set to a new stage, its time will be halted. The new instance´s LST is resumed and the GST is synchronized to that. This topic is the most important, because it influences the sequence of the simulation and ensures coherency to real-time conditions of a job set. The reference clock of all Avatars, Nodes and Elysion is the LST of the current processed job.

2. **Message passing** In a message passing system, there are two counterparts, the sender and the receiver of messages. The `MessageProcessor` compares their LSTs (+ latency time) and synchronizes them to the latest, if they differ.

3. **Performing Titan** Titan´s reference clock is always the GST, during its scheduling and reservation turns it sends messages toward Avatars representing resource owners. As mentioned before, their associated Nodes have to be switched, because only one Node can be simulated at once. Due to the issue, that this will never occur in reality, all simulation times are on hold in the meantime and are resumed when it is performed.

4. **Finishing the simulation** The GST is synchronized to the defined final-time of the simulation when all jobs of the working-set are processed and have reached a final stage.

4.2.2.4 Event Handling

Events are incidents, that affect the simulation process instantly. These may be thrown by different entities, i.e. Avatars, Titan and Elysion. Thus their appearance differs and have to be managed variably. Each part of Elysion can be an erector of events. Thus major parts of the implementation of the simulation processor are event-driven. There are several classes to provide these functionalities: **EventCollector**, **EventProcessor** and **BoundedBuffer**. Their roles are illustrated in Figure 4.10.

An event is represented by the `EventToken` class (see Figure 4.1) containing all necessary fields to comply event-handling. To figure out the type of an event, it has got an integer typed code, which is defined in the `EventTokenInterface`. There each number has an associated Java static final defier, which is more comfortable for programmers. A list of current processed event codes is given in the enumeration below:

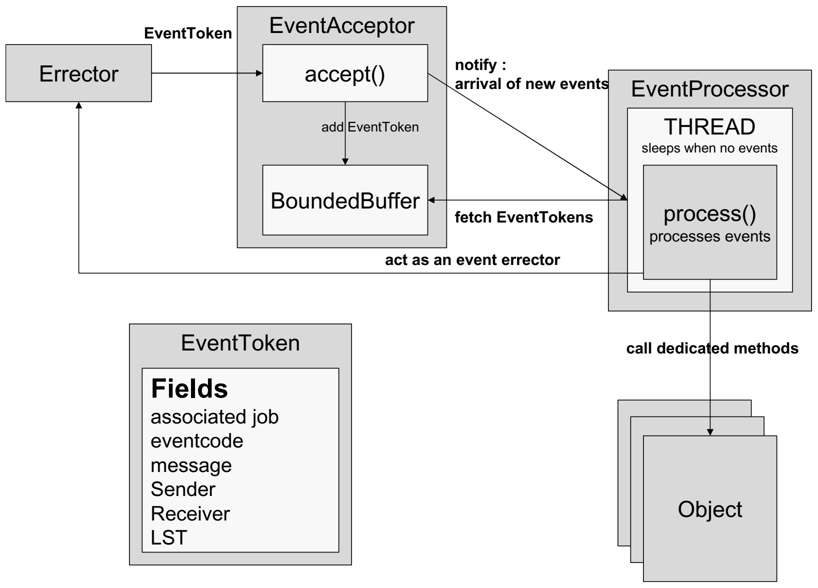


Figure 4.10: Event processing.

1. **SEND_MESSAGE** Thrown by the `Connector` class to invoke the `MessageProcessor` to process a sent message.
2. **STRT_NODE** The `NodeDispatcher` is wanted to start a Node from its queue.
3. **STOP_NODE** The `NodeDispatcher` is wanted to start a Node from its queue.
4. **AVT_IDLESIGNAL** Thrown by the `Connector` class to indicate that a running Avatar instance became idle and can be shut down. This is usual done by Bob-typed Avatars, when the actual job is in `EXECUTIONSTAGE`.
5. **AVT_ACQUIRESIMTIME** Thrown by the `Connector` class to indicate that a running Avatar is requesting simulation time for future.
6. **SCHD_NEXTJOB** Invokes the `JobProcessor` to select a new job for simulation and starts its processing. If the current job is in a non-preemption state, this event has no effect.
7. **JOB_RESFINISHED** Thrown by the reservation client of `Titan` appointing a succeeded reservation. The event is passed to the `JobProcessor` to advance the job stage, that it is ready for execution.

8. **JOB_RESFINISHEDFAILED** The opposite case of the previous event, then the job will be terminated by the `JobProcessor`.
9. **PROC_JOB** The `JobProcessor` will continue processing its actual job without scheduling a new one.
10. **JOB_SCHFINISHED** Thrown by `Titan` appointing a succeeded schedule, which is stored in job´s associated `JobEntity`. The event is passed to the `JobProcessor` to advance the job stage, that it is ready for reservation.
11. **JOB_SCHFINISHEDFAILED** The opposite case of the previous event, the job will be terminated then by the `JobProcessor`.
12. **JOB_FINISH** Thrown by either the `JobProcessor` or the EventProcessor itself as a consequence of another event
    (for e.g. **JOB_SCHFINISHEDFAILED**). It will end the simulation of the current job and invokes a new job schedule.
13. **NO_JOBSLEFT** Thrown by the `JobProcessor` when all jobs have been simulated completely. The simulation turn will end.
14. **AVT_CONNSIGNAL** Thrown by started Avatars to indicate their connectivity to Elysion.
15. **AVT_STOPSIGNALACK** Thrown by Avatars to be stopped indicating their acknowledge.
16. **AVT_STARTSIGNALACK** Thrown by started Avatars to indicate their readiness to start their execution.
17. **STOP_SIGNAL** Thrown by the `Connector` to invoke generation of notification messages for the current Node, to inform the associated Avatars to cease their activities, that they can be terminated without harm.

As illustrated in Figure 4.10, any object can be an event-erector. This is done by generating an `EventToken` instance which is to be passed to the `EventCollector` by invoking its `accept()`-method. It will add the transmitted `EventToken` to its internal event-buffer (`BoundedBuffer`) and will notify the `EventProcessor`, that the buffer was filled, which wakes up and processes each event in the buffer until it is empty. Then it falls asleep to save CPU time. For each event the `EventProcessor` forks a new thread, to keep liveness in the whole process.

4.2.3 Evaluation Facility

To gain conclusions about a simulation turn, empiric data must be generated and provided. These can be obtained by tracing the state of a simulation, which can be represented by integers, strings or whole objects. A typical use-case is to keep trace about the load of CPUs in a defined period. The `evaluation` package is a common architecture to comply those assignments. Its primer goal is to trace values of object instances.

4.2.3.1 Components

The design of this package is held common and can easily be adapted to other software project. The implementation of the evaluation facility is shown in Figure 4.11 is structured into four major parts:

1. **LogObjects** Any Objects thats attributes or values are to be traced must extend the `LogObject` class and becomes itself a `LogObject`. Then all needed functionalities are implemented. A complete guide about logging is given in Section 4.2.3.2.

2. **LogProperties** The abstract class `LogProperty` represents a single attribute to be traced. The inherited classes `LogPropertyInteger`, `LogPropertyFloat`, `LogPropertyString` and `LogPropertyObject` can be instantiated and must comply to the data-type of the attribute. `LogEntities` (see Figure 4.12) are snapshot of values, detailed information can be found in Section 4.2.3.2.

3. **LogBooks** `LogObjects` can be grouped to a `LogBook` object. Each one must contain at least on objects. The top is the `LogBookLibrary`, a container of `LogBooks`. This hierarchical structure is given to make it easier for display issues in GUIs, because it builds up an instance of an Java Swing `JTree`. So that user can select any object to evaluate its traces. The `LogBookLibrary` is quite proprietary and must be modified for other software projects, because it initializes the log environment for Elysion, too.

One class has not been mentioned yet, the `LogUnit`. This one is the predecessor of the current implementation, its primary intention was to log everything of a simulation turn into a text file located in the working path of the simulation turn. Due to the circumstance that the simulation package became a graphical user interface (see Section 4.5), which interacts with the current implementation, its actual assignment is to produce **workloads** of all jobs being processed by Elysion. The **JobEntity** instances, which each represents a single job, contains all information to produce workloads. These are kept an managed by the `JobProcessor`. In the text file, each line is the result of a job process and contains following fields:

1. **ID** The ID of the job given by Elysion.
2. **Prototype** If a workload input (see Section 6.6.1) was provided, the number correspondents to the ID of the used template job according to the sequence in the input XML-file.
3. **Exitstage** The last stage that a job has reached, after its processing has terminated (see Figure 4.8). This is an integer value, and its associated stage can be reviewed either in the graphical user interface or in the `JobProcessor` class.
4. **NodesReq** The total number of different Nodes, that have been activated during job processing (scheduling and reservation).
5. **Reqcnt** Total number of requests to Nodes (scheduling and reservation).
6. **Submission** The LST when the job was committed to Titan (long value). This value equals to zero, if something went wrong before.
7. **Dispatch** The LST when Titan has finished scheduling (long value). This value equals to zero, if something went wrong before.
8. **Exec** The LST when the job´s stage reached `EXECUTIONSTAGE` (long value).This value equals to zero, if something before went wrong.
9. **Term** The LST when the job´s processing was finished (long value). This value equals to zero, if something went wrong before.

An example workload output is given below:

```
;CreationTime:12.02.2004 09:04:17
;SimulationName:SimulationsTest
;ProcessingComputer:host12.user12
;WorkingDirectory:c:\temp\gridsched2\simulation\
;SimulationRMIPort:2099
;SimulationLDAPHost:127.0.0.1
;SimulationSQLDB:gridsched
;LineFormat: ID Prototype ExitStage NodesReq ReqCnt
;            Submission Dispatch Exec Term
0 0 5 3 11 975668379648 975668380429 975668510220 975668910220
```

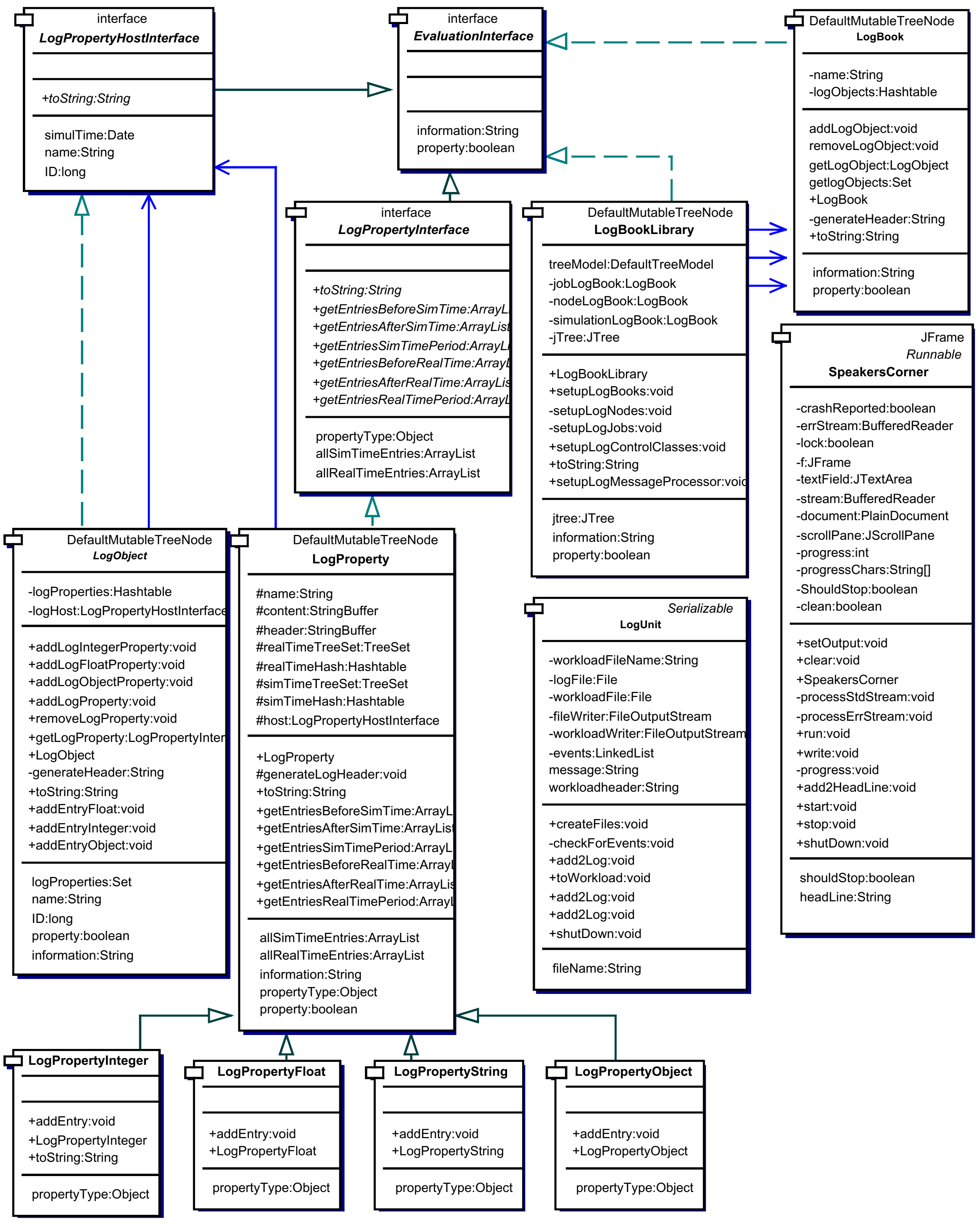


Figure 4.11: Evaluation facility classes.

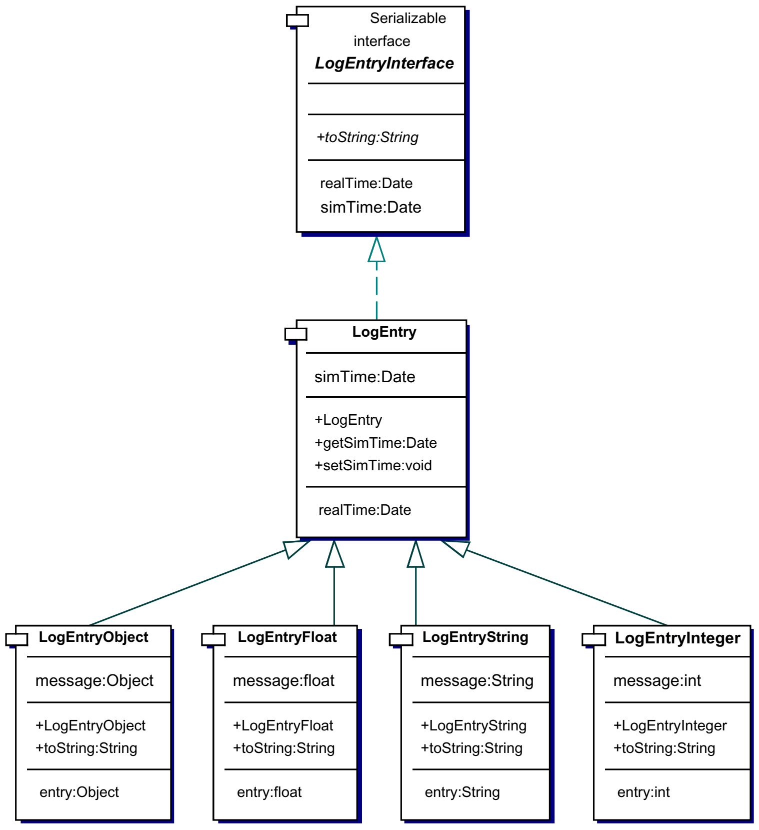


Figure 4.12: LogEntry classes.

4.2.3.2 Implementation Guide

This sections is intended as a brief manual for using the log mechanisms of the evaluation facility. First of all it should be deliberated which objects should be traced. Then all desired class definition must extend the `LogObject` class. Now it is able to obtain `LogProperties`. It is important to set the correct data-type (see Section 4.2.3.1). Due to the ability to attach more than one property, each one is associated with an hash-key.

An example code of is given below to add a integer-typed property to a new `LogObject`:

```
...
static final string DESC="Node Count Trace";
static final string HASH="nodecount";
int value=0;
LogClass logObject=new LogClass(); // extends LogObject class
LogPropertyInteger property=new LogPropertyInteger();
logObject.addLogIntegerProperty(DESC,HASH);
...
```

These properties are not directly coupled with attributes or values and must be managed by hand. That means, all attribute´s changes must be set to their dedicated properties manually. Properties can be filled with `LogEntry` Objects, these are typed too. So there are `LogEntryObject`, `LogEntryString` and so on (see Figure 4.12), they represent snapshots of an attribute´s change. These snapshot have to invoke manually by calling methods. To set a new entry, each property provides an `addEntry()` method (see Figure 4.11). The LogObject provides the same mechanisms to shorten implementation. When a `LogEntry` instance is being created, the real-time and LST is attached to it. Furthermore each entry carries a message

Class	Properties
Node	activation trace
JobEntity	node requests (for schedule and reservation), stage, schedules
MessageProcessor	processes messages
JobProcessor	jobs left, Node load
EventProcessor	processes events

Table 4.1: Property traces.

that describes the state, respectively the content of a value or an object that is snapshot. So it is even possible to trace the whole state of all program´s objects over the time for e.g. .

To continue the example code:

```
...
value=1;
logObject.addEntryInteger(HASH,value);
value=2;
logObject.addEntryInteger(HASH,value);
...
```

To receive the traces of a property for evaluation in programs, the class provides several methods, `getEntriesAfterSimTime()` or `getEntriesBeforeRealTime()` for instance.

Table 4.1 lists properties, that are traced by the current implementation of the simulation program. These can be viewed in the graphical user interface (see Section 4.5).

To group on ore more `LogObjects` to a `LogBook`, its `addLogObject()` method are called by submitting a hash value and the objects reference. This is analogue to the addition of `LogProperties` to `LogObjects`. To reference log entities in general, their hash values can be applied to their hosted object instances. As shown above, it is quite simple to implement further properties to be logged and traced.

4.3 Integration of GridSched Services

To make an existing GridSched service work with the simulation environment, it must include an instance of the `SimulationProxy` class as introduced in Section 4.2. It provides a common interface to communicate with the simulation processor. From the service´s point of view, it is applicable in a very transparent way. Thus all services must distinguish between a 'real mode', where the `SimulationProxy` is not used, and a 'simulation mode' by implementing it. This is to supposed to do via command line options. When the simulation is starting an Avatar process, the command line switches '`-SimHub:[network address]`' and '`-SimID:[virtual network address]`' are committed to the executables. The first one determines the host machine, where the simulation program is running, the latter one sets an unique identifier, equal to the virtual network address (see Section 4.2.2.1), that each Avatar should own. These information are necessary to establish an RMI connection to the simulator. If these options are not set, then the Avatar shall run in 'real mode'.

4.3.0.3 GridSched Service Requirements

To define the requirements towards a service, in order to be controllable by the simulation program, all these conditions below must be met:

1. Two modes of operation, the 'normal mode' and the 'simulation mode'.
2. All components of the service must be interruptible in their execution.
3. The whole service must be able to save its current state on disk to claim a later resume of execution.

4. The `SimulationProxy` class must be applied.

The next section illustrated the assignments of the `SimulationProxy` class being integrated in a service.

4.3.0.4 SimulationProxy Functionalities

An overview is given below, what functions are covered by the `SimulationProxy` class:

- **Control over the host program** It is evident, that an interface towards the host program must exist, to assume power over it. This is realized by the `AvatarControl` interface. Thus at least one class must implement it, but at much as total control over the whole program is assured. The method `start()` indicates the component, that it is allowed to execute, the opposite `stop()` method signals, that it must interrupt its execution to gain a safe state, where it can be interrupted without any data loss. Before the `SimulationProxy` object can access any of these methods of an entire component, it must be registered to it by invoking the `register()` method. The mechanism is simple by providing the `AvatarControl` interface reference of the object to register.
- **Persistence** To comply this special requirement as described in the text above, the `AvatarControl` interface includes a `read()` and a `write()` method. The first one may only be called, before the `start()` method has been called, the latter one, after the `stop()` method has been invoked. Their meaning should be self-evident. From the implementation´s point of view, these functionalities are realized by providing Java `ObjectStreams`. An advantage of this mechanism is that the `SimulationProxy` does not need to take care about the internal data-structures of the components.
- **Absorption of all communication package functionalities** The `RMIInterface` is also implemented by the `SimulationProxy`, so that service components can use the communication in their familiar way. But is it to be kept in mind, that virtual network addresses must be used (see Section 4.2.2.1). Even operations on remote directories are complied by implementing the `JNDIInterface`, the mechanism is equal to the reality mode, but the simulation environment is using its own directory service, hosted by via the `StaticResourceLibrary` class, to prevent interdependencies. All service components should access their `RMIProxy` and `JNDIProxy` via the interfaces `RMIInterface` and `JNDIInterface` and not via the concrete implementations to gain transparency between simulation and real mode.
- **Time service** It provides a `getTime()` method to deliver the actual local simulation time. The usage of this is most important for the simulation mode, because the LST usually differs from the real-time.
- **Simulation event processing** As indicated in Section 4.2.2.4, Avatars can process events by sending or receiving messages via the `RMIInterface`. Furthermore they can signal that they have become idle, by invoking the `setIdle()` method, and can be interrupted. This event form only affects the simulation environment, when all Avatars of an active Node became idle. Another application, as used by Bob in the `READYTORESERVESTAGE`, is to claim simulation time for predictable incidents by calling the `throwEvent()` method, where a time stamp is passed to determine the occurrence of these. And also a description string, to log and identify, when the event is processed. When an Avatar got the acquired simulation time, it must call itself the `setIdle()` method from the `SimulationProxy` to signal that it requests its termination, otherwise it would gain simulation-time infinitely. The `SimulationProxy` avails itself of these events to negotiate with the simulation processor, i.e. waiting for stop signals or something similar, by using the form of `EventTokens`. A more detailed point of view of their usage is explained in Section 4.3.0.6, where the processes, that happen inside of Avatars, when a Node context switch is scheduled.

4.3.0.5 Implementing the SimulationProxy in Services

This section is a short instruction, how to include the `SimulationProxy` into a host program, to make it compatible with the simulation program.

1. Modify the code, that both modes of execution are provided (see Section 4.3).
2. Parse the command line for the options '`-SimHub:`' and '`-SimID:`'.
3. Instantiate a `SimulationProxy` object, pass the command line arguments and the count of components, that will be registered and assign its reference to `RMIInterface` and `JNDIInterface` attributes both, if necessary. Make it visible to all objects, that access its functionalities.
4. Complete all methods defined by the `AvatarControl` interface for all classes implementing the interface.
5. Perform the familiar initialization activities. Keep in mind, that no component should start working before the `SimulationProxy` invoked the `start()` methods.
6. Register all classes to the SimulationProxy object by invoking the `register()` method.

4.3.0.6 Avatar Initialization and Termination

This section is dedicated to show all processes, that happen inside of Avatars, when the simulation processor is going to start or stop their associated Nodes. The Avatar initialization ties in with Section 4.3.0.5, so the focus is set on the `SimulationProxy` during these phases.

- **Initialization** The `SimulationProxy´`s constructor performs several operations to bind itself to the simulation environment (see Figure 4.13). First, it looks up the `Connector` RMI instance of the simulation running on the given host provided by command-line options. Then it requests the actual local simulation time, and sends out an `EventToken` for greeting issues, to signal the `Connector` connectivity. The next assignment is to clarify, if the service has already been simulated, if it is true, then the information about an existing state-file, that has been saved before, is received. Afterwards an internal thread is started to listen for stop signals.

  When these steps are finished, the components must register itself to the `SimulationProxy` by calling the `register()` method (see Figure 4.13). In each call, the count of registered components is incremented. If all entities has been registered, the `SimulationProxy` resumes activity and checks, if a state file exists. If not, all `start()` methods, provided by the `AvatarControl` interface, are invoked. Otherwise, if the condition was true, the state file is loaded and piped into an `ObjectInputStream` and passed to all registered components by calling their `setState()` methods before. Afterwards an `AVT_STARTSIGNALACK` event is thrown towards the simulation environment. Finally it waits for a '`go event`' from the processor indicating, that the host can operate. This command is passed by calling the `start()` methods of all registered components.

- **Termination** The listener thread for 'stop running' events introduced in the last point loops until such a event is thrown, then it calls the internal `stopAvatar()` method to begin all necessary steps to shut down the host program (see Figure 4.14). This method calls all `stop()` methods from the registered components, invoking their operational termination. These calls should only set internal flags to prevent blocking of the `SimulationProxy`, then it loops until all calls of the `getRunningFlag()` method result negatively. The next step is the preparation of the state file, whose filename is among the others composed of the virtual network address. The file is typed as '.hst', which stands for 'hades state file'. This file is now generated and combined with an `ObjectOutputStream`, which is passed to all registered components by calling their `getState()` methods (see Figure 4.14). Finally a `EventToken`, containing all information about the written state file, is sent to the simulation processor, to signal, that the Avatar has been shut down and its process can be terminated.

4.4 GridSched Component Alignments

This section delves into the alignments towards the simulation, that have been done to make them compatible. The GridSched services Bob and DataManager are modified as described in Section 4.3.

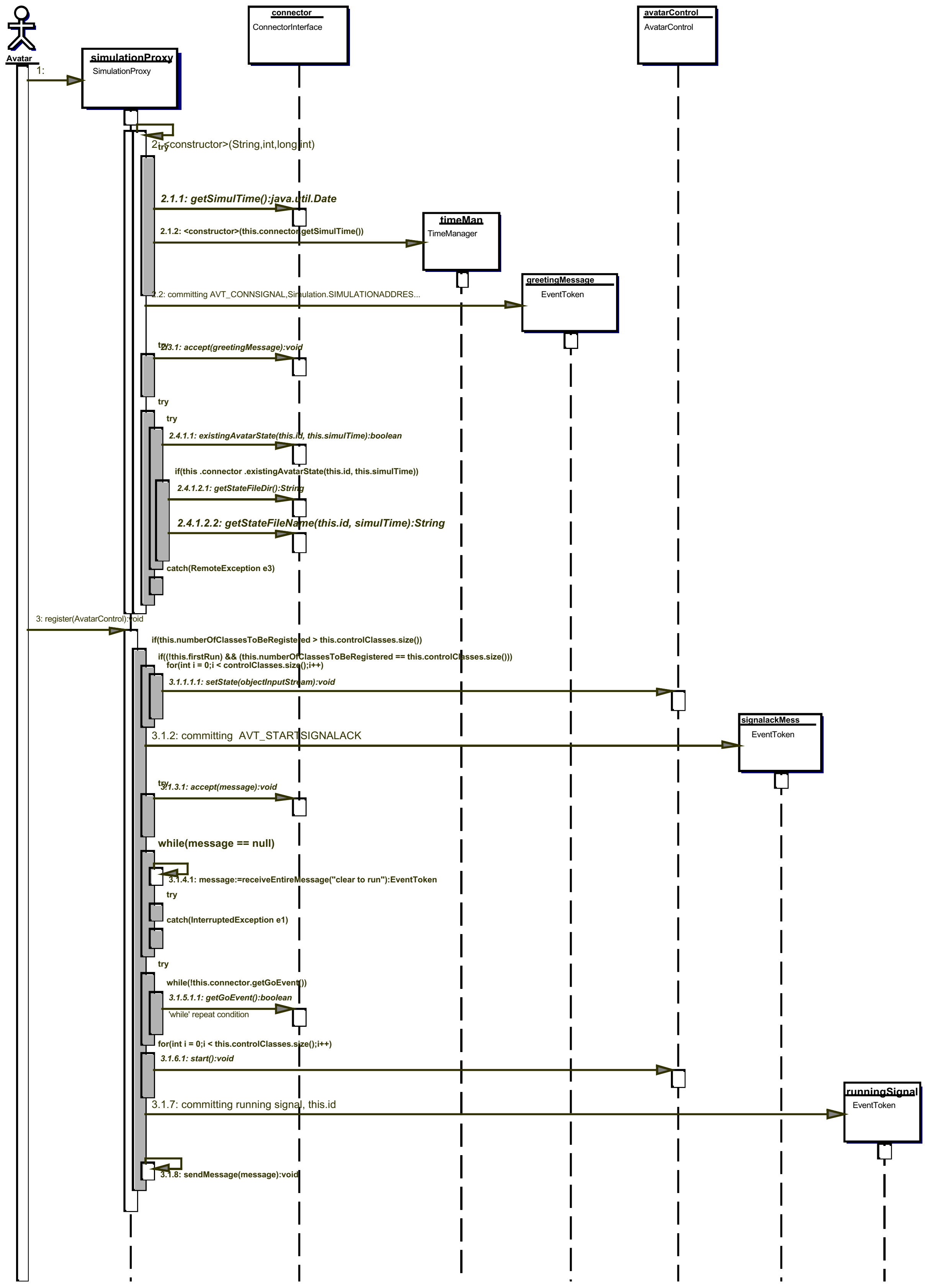

Avatar
simulationProxy
SimulationProxy
connector
ConnectorInterface
avatarControl
AvatarControl
1:
2: <constructor>(String,int,long,int)
try
2.1.1: getSimulTime():java.util.Date
2.1.2: <constructor>(this.connector.getSimulTime())
timeMan
TimeManager
greetingMessage
EventToken
2.2: committing AVT_CONNSIGNAL,Simulation.SIMULATIONADDRES...
try
2.3.1: accept(greetingMessage):void
try
try
2.4.1.1: existingAvatarState(this.id, this.simulTime):boolean
if(this .connector .existingAvatarState(this.id, this.simulTime))
2.4.1.2.1: getStateFileDir():String
2.4.1.2.2: getStateFileName(this.id, simulTime):String
catch(RemoteException e3)
3: register(AvatarControl):void
if(this.numberOfClassesToBeRegistered > this.controlClasses.size())
if((!this.firstRun) && (this.numberOfClassesToBeRegistered == this.controlClasses.size()))
for(int i = 0;i < controlClasses.size();i++)
3.1.1.1.1: setState(objectInputStream):void
signalackMess
EventToken
3.1.2: committing AVT_STARTSIGNALACK
try
3.1.3.1: accept(message):void
while(message == null)
3.1.4.1: message:=receiveEntireMessage("clear to run"):EventToken
try
catch(InterruptedException e1)
try
while(!this.connector.getGoEvent())
3.1.5.1.1: getGoEvent():boolean
'while' repeat condition
for(int i = 0;i < this.controlClasses.size();i++)
3.1.6.1: start():void
runningSignal
EventToken
3.1.7: committing running signal, this.id
3.1.8: sendMessage(message):void

Figure 4.13: Avatar initialization.

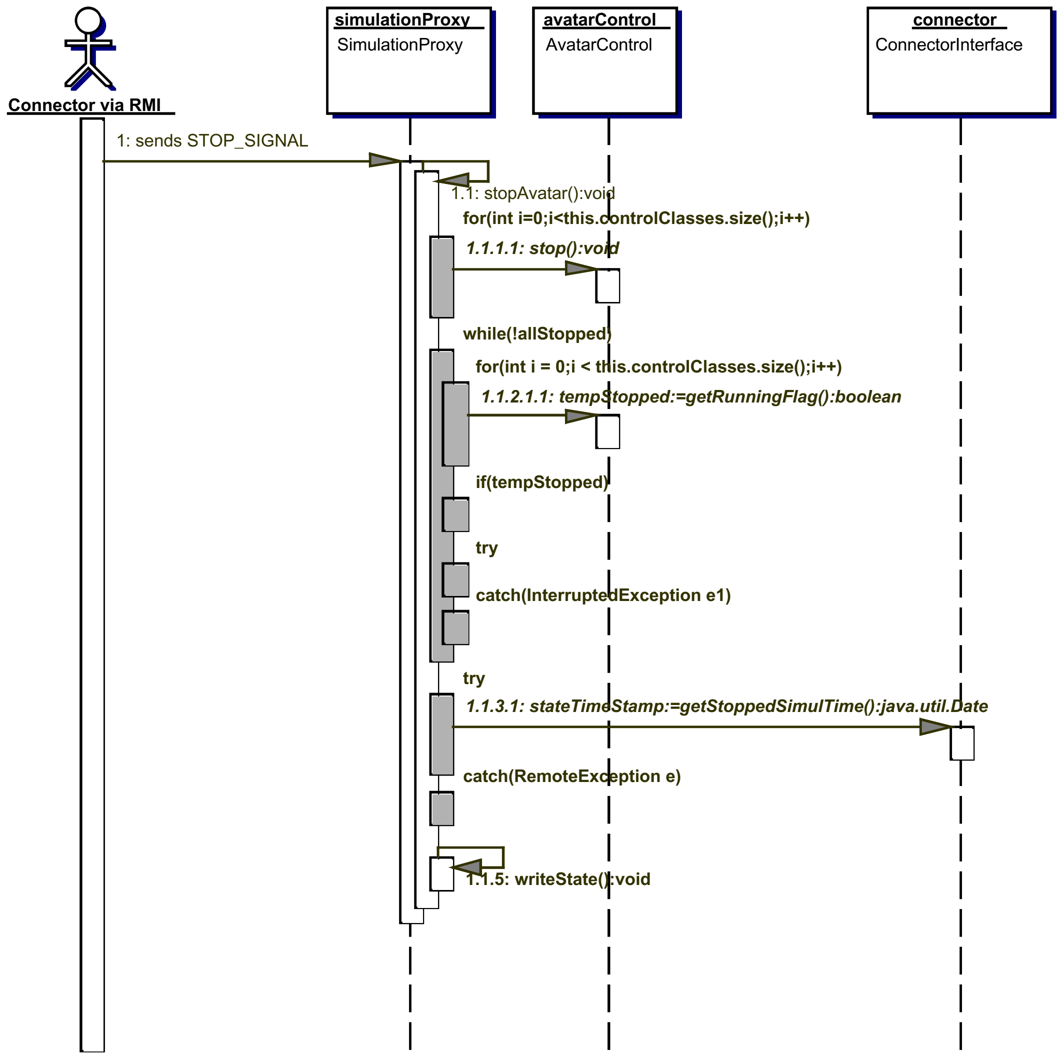
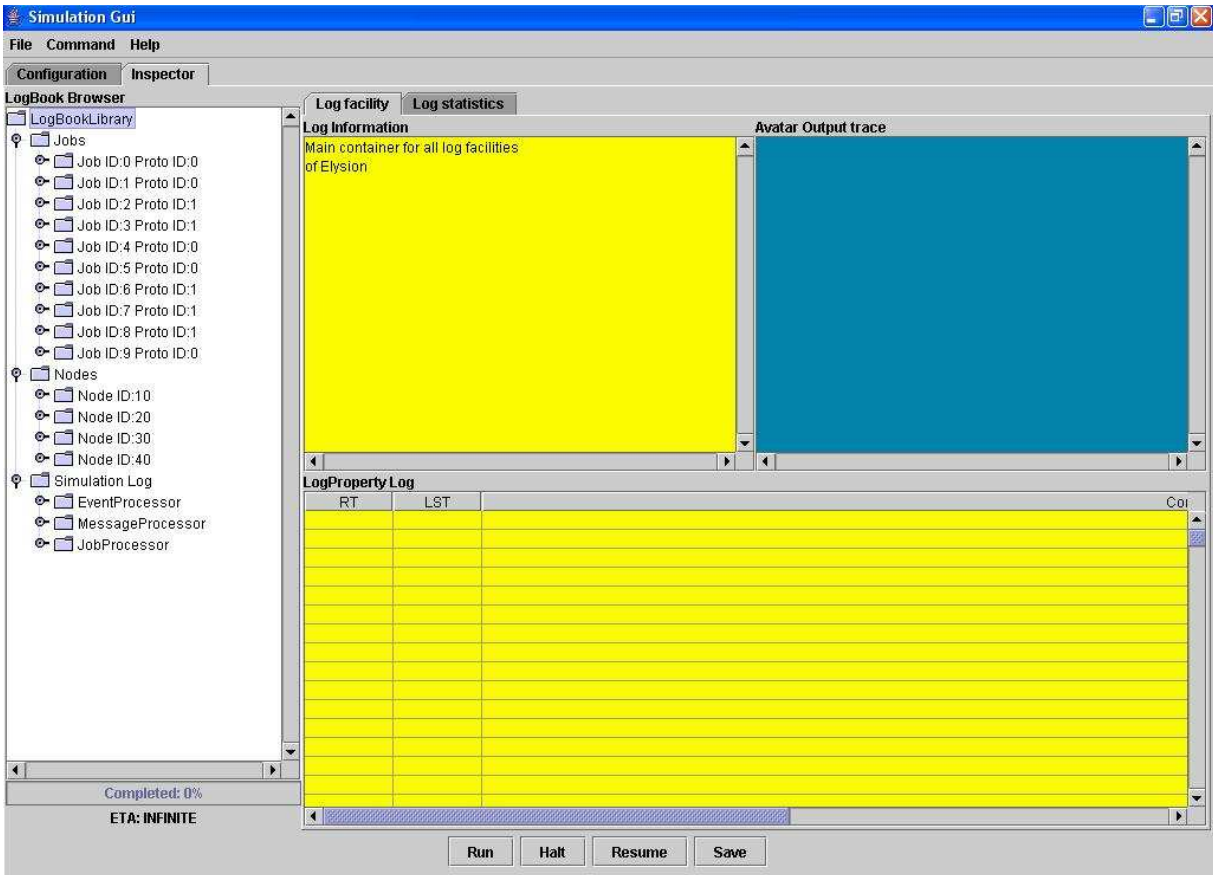


Figure 4.14: Avatar stop sequence.

They only provide one component to register to the SimulationProxy, their main classes. Due to the circumstance that Bob manages resources, it makes no sense to run it without holding any. Thus, the SimulationProxy provides a mechanism to fill it with some initial resources, as defined in the XML document of the simulation scene input file. This is only done, when the entire Bob is started for the first time, because its persistence function will save them, too. Okeanos stores their descriptions in the StaticResourceLibrary class publishing them to a remote directory server. As mentioned before Titan became part of the simulation processor, so the entire 'simulation mode' is not invoked by command-line but rather via a special constructor call. Detailed information how Titan is involved in the simulation is described in Section 4.2.2.2. Both, Titan and Bob, have access to a log() method in their main classes to provide important information about their activities to the log facility of the simulation. Another effect of the inclusion of Titan into the processor is that, its message traffic does not leave it via RMI mechanism. So the communication package also provides an simulation mode, where it modifies the RMI functions of the RMIProxy to map it to the virtual network of the simulation processor. But this modifications affect only Titan.

4.5 Simulation GUI

The Simulation GUI offers an availability to build up a simulation, run a simulation, view the output and the state during a simulation run and to analyze the result after running a simulation. A description of how these aspects are realized is following.



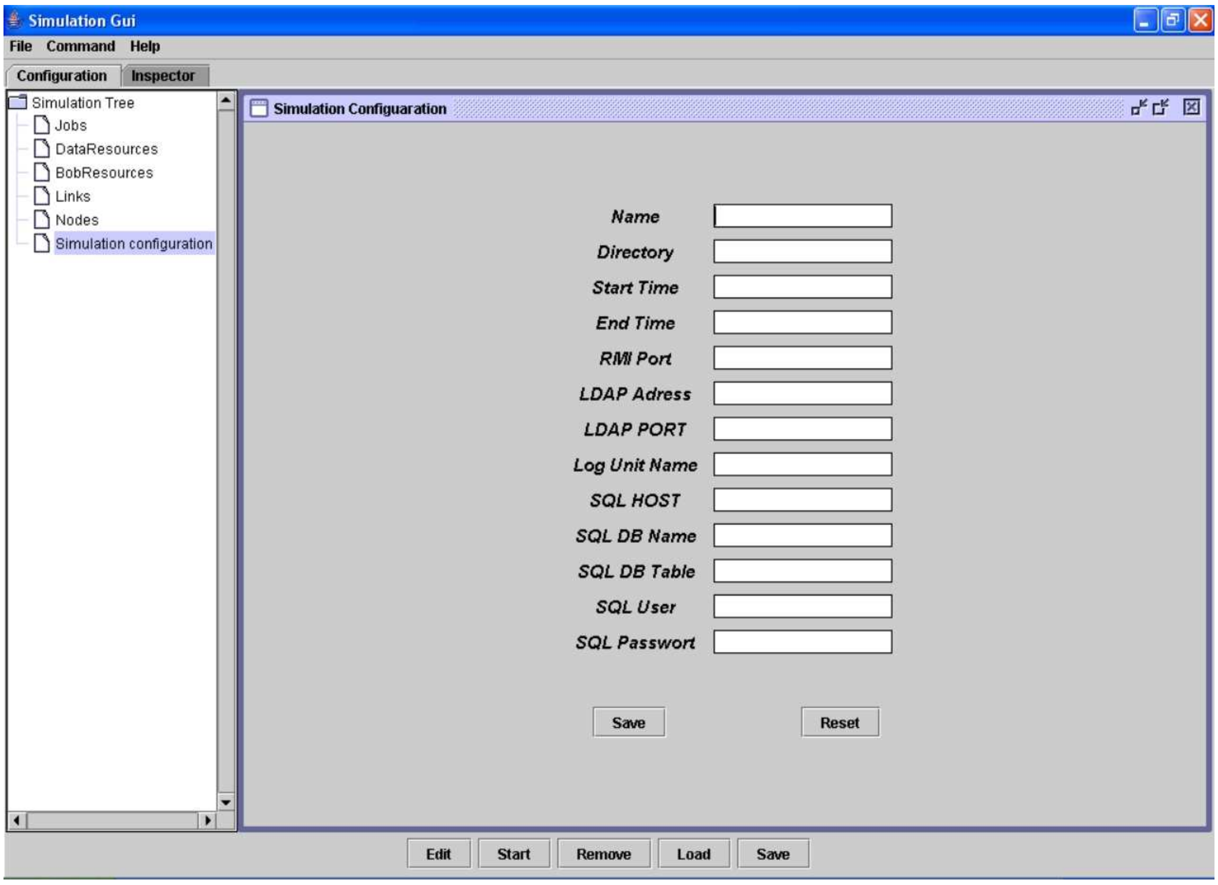
Main container for all log facilities
of Elysion

Figure 4.15: Overview of the Simulation GUI.

The Simulation GUI is divided in two main parts. One part consists of a configuration tab in which a simulation can be build up by entering Nodes, Bob and Data Resources, jobs and general configuration settings (see Figure 4.16). The other part is the inspector tab which displays the state and the results of the simulation.

Configuration Tab The configuration tab contains on the left side a tree which displays all editable components. The right side displays the component which should be edited. There are several buttons available in the configuration tab.

- **Edit** After selecting a component in the tree, this component is editable after clicking the edit button. After entering all necessary values and clicking the save button a corresponding JAVA object will be created.
- **Start** When all needed components are entered the simulation can be started by clicking this button.
- **Remove** The selected component will be removed by clicking this button.
- **Save** Saves a whole simulation configuration with all created objects.
- **Load** Loads a whole simulation configuration with all saved objects.



Name	
Directory	
Start Time	
End Time	
RMI Port	
LDAP Adress	
LDAP PORT	
Log Unit Name	
SQL HOST	
SQL DB Name	
SQL DB Table	
SQL User	
SQL Passwort	

Figure 4.16: Simulation configuration tab.

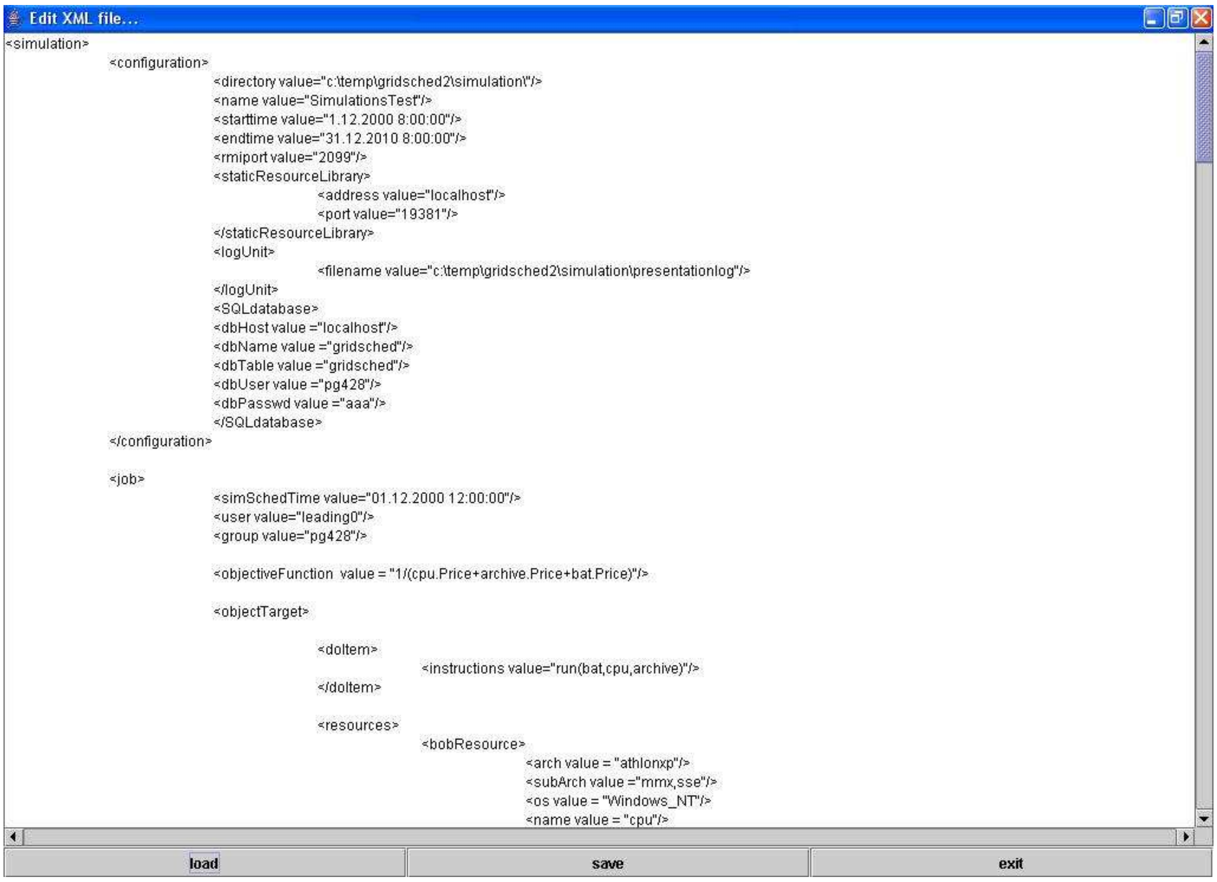
As mentioned before it is possible to enter the configuration of a simulation via the GUI but it there is also the possibility to use an XML-document to build up a simulation scene. Such an XML-document describes the settings of the simulation. Further details about the XML-documents are given in Section 6.6.1. On top of this a text editor is integrated in the simulation GUI to offer the opportunity to change the values in the XML-file (see also Figure 4.17).

After building a simulation environment a simulation run can be started by clicking the run button in the inspector part of the simulation GUI. It is also possible to halt a simulation and resume it later. After stopping a simulation it can be saved to a file. It is of course possible to reload this file and to resume the simulation turn later. Some technical details about saving a simulation are given in Section 4.2.1.

Inspector Tab The inspector tab itself is divided in two more tabs, one called log facility which shows information about the state and the result of a simulation and a tab called log statistics which allows to show charts concerning selected parts of an simulation. The inspector tab offers on the left side a so called logbook tree in which general information about the simulation and about jobs and nodes are displayed. During a simulation these values are updated so that the state of the simulation in general, for example how many jobs are already simulated, can be examined as well as the state of a selected part of the simulation, for example a node and the requests send to this node. A more detailed explanation is given in Section 4.2.3.2.

The log facility tab shows on the right side several widgets which are displaying the output that Avatars produce, a log information widget and a widget in which all events could be displayed.

The log statistics tab offers a window in which selected properties can be added via drag and drop. It is possible to choose the graphical representation of these properties, e.g. lines, bars or pie charts which are then displayed below, for example see Figure 4.19 where the finishing times of jobs are represented in a



```
<simulation>
        <configuration>
                <directory value="c:\temp\gridsched2\simulation\"/>
                <name value="SimulationsTest"/>
                <starttime value="1.12.2000 8:00:00"/>
                <endtime value="31.12.2010 8:00:00"/>
                <rmiport value="2099"/>
                <staticResourceLibrary>
                        <address value="localhost"/>
                        <port value="19381"/>
                </staticResourceLibrary>
                <logUnit>
                        <filename value="c:\temp\gridsched2\simulation\presentationlog"/>
                </logUnit>
                <SQLdatabase>
                <dbHost value ="localhost"/>
                <dbName value ="gridsched"/>
                <dbTable value ="gridsched"/>
                <dbUser value ="pg428"/>
                <dbPasswd value ="aaa"/>
                </SQLdatabase>
        </configuration>

        <job>
                <simSchedTime value="01.12.2000 12:00:00"/>
                <user value="leading0"/>
                <group value="pg428"/>

                <objectiveFunction  value = "1/(cpu.Price+archive.Price+bat.Price)"/>

                <objectTarget>

                        <doItem>
                                <instructions value="run(bat,cpu,archive)"/>
                        </doItem>

                        <resources>
                                <bobResource>
                                        <arch value = "athlonxp"/>
                                        <subArch value ="mmx,sse"/>
                                        <os value = "Windows_NT"/>
                                        <name value = "cpu"/>
```

Figure 4.17: Simulation GUI XML Editor.

diagram. It is possible to choose between a real time and a simulation time axis. In this case a real time axis was chosen to see how long it took to simulate each of these jobs.

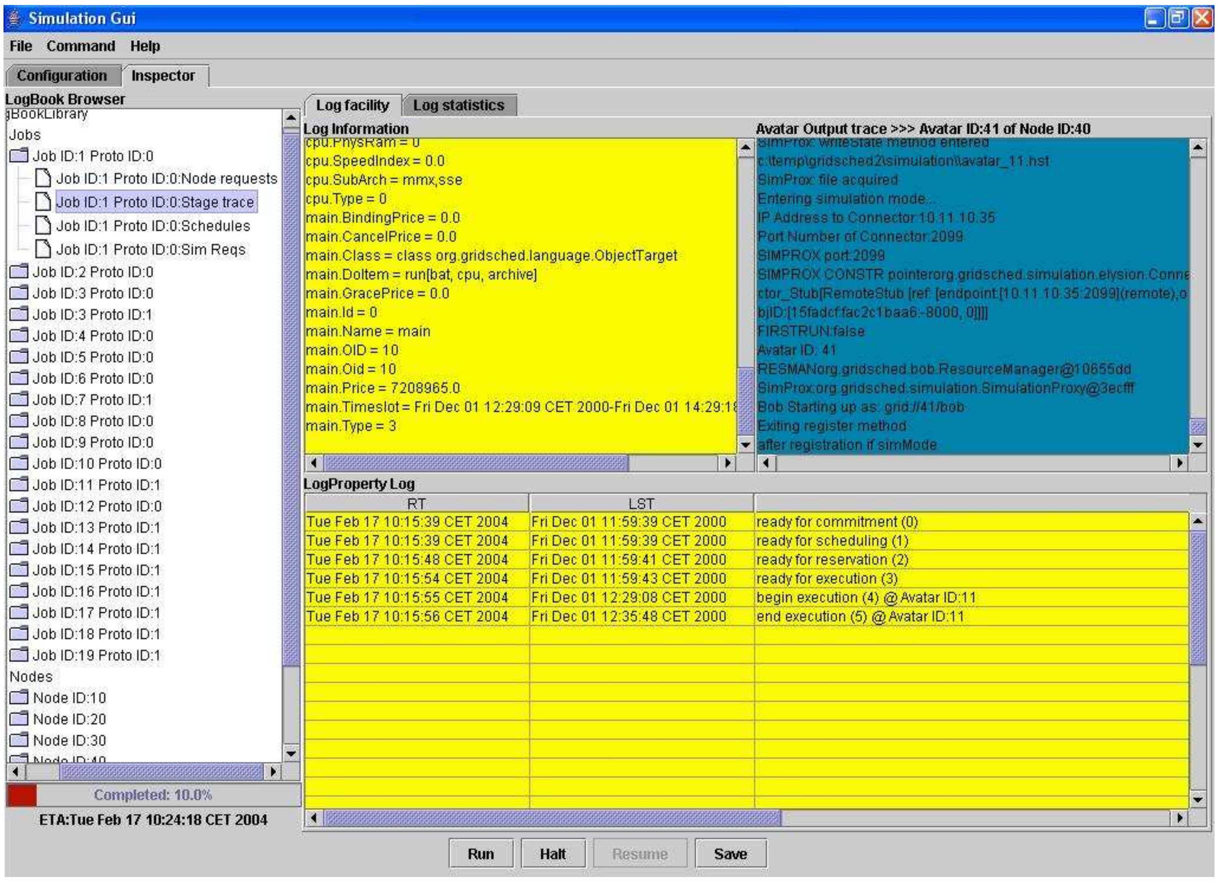
Another example for a graphical analysis is given in Figure 4.20. In that graphic it is possible to see how often a node was requested to execute a job. In this case the `DataManager` on node 20 was requested six times to execute the job.

A bar which shows the progress of the simulation is placed under the log browser. This bar shows a percentage which states how much of the simulation has already been completed. On top of this the bar changes its color from red over orange to green while the simulation is proceeding. Another feature this status control element offers, is to estimate the completion time of the simulation. This is done by multiplying the average time the already simulated jobs needed to execute with the number of jobs left and adding this result to the actual time to get approximately the time by which the simulation will be finished.

4.6 Summary

The simulator offers an opportunity to simulate the GridSched middleware. It it possible to simulate jobs like they would be executed in the GridSched environment. All components of GridSched are working in the same way as they do in normal mode just a few adaptations were necessary (see Section 4.3 for details). Therefore it is possible to examine the real behaviour of the GridSched middleware by using the simulator. It was possible to detect a lot of failures and problems of GridSched in that way during the development of the project.

A lot of functionalities were added during the second term of the project. The simulator is able to simulate an arbitrary number of jobs which can be entered either by an XML-document (see Section 6.6.1) or via



LogBook Browser

```
gBookLibrary
Jobs
  Job ID:1 Proto ID:0
    Job ID:1 Proto ID:0:Node requests
    Job ID:1 Proto ID:0:Stage trace
    Job ID:1 Proto ID:0:Schedules
    Job ID:1 Proto ID:0:Sim Reqs
  Job ID:2 Proto ID:0
  Job ID:3 Proto ID:0
  Job ID:3 Proto ID:1
  Job ID:4 Proto ID:0
  Job ID:5 Proto ID:0
  Job ID:6 Proto ID:0
  Job ID:7 Proto ID:1
  Job ID:8 Proto ID:0
  Job ID:9 Proto ID:0
  Job ID:10 Proto ID:0
  Job ID:11 Proto ID:1
  Job ID:12 Proto ID:0
  Job ID:13 Proto ID:1
  Job ID:14 Proto ID:1
  Job ID:15 Proto ID:1
  Job ID:16 Proto ID:1
  Job ID:17 Proto ID:1
  Job ID:18 Proto ID:1
  Job ID:19 Proto ID:1
Nodes
  Node ID:10
  Node ID:20
  Node ID:30
  Node ID:40
```

ETA:Tue Feb 17 10:24:18 CET 2004

Log Information

```
cpu.PhysRam = 0
cpu.SpeedIndex = 0.0
cpu.SubArch = mmx,sse
cpu.Type = 0
main.BindingPrice = 0.0
main.CancelPrice = 0.0
main.Class = class org.gridsched.language.ObjectTarget
main.DoItem = run[bat, cpu, archive]
main.GracePrice = 0.0
main.Id = 0
main.Name = main
main.OID = 10
main.Oid = 10
main.Price = 7208965.0
main.Timeslot = Fri Dec 01 12:29:09 CET 2000-Fri Dec 01 14:29:1
main.Type = 3
```

Avatar Output trace >>> Avatar ID:41 of Node ID:40

```
SimProx: writeState method entered
c:\temp\gridsched2\simulation\\avatar_11.hst
SimProx: file acquired
Entering simulation mode...
IP Address to Connector:10.11.10.35
Port Number of Connector:2099
SIMPROX port:2099
SIMPROX CONSTR pointerorg.gridsched.simulation.elysion.Conne
ctor_Stub[RemoteStub [ref: [endpoint:[10.11.10.35:2099](remote),o
bjID:[15fadcf:fac2c1baa6:-8000, 0]]]]
FIRSTRUN:false
Avatar ID: 41
RESMANorg.gridsched.bob.ResourceManager@10655dd
SimProx:org.gridsched.simulation.SimulationProxy@3ecfff
Bob Starting up as: grid://41/bob
Exiting register method
after registration if simMode
```

LogProperty Log

RT	LST	
Tue Feb 17 10:15:39 CET 2004	Fri Dec 01 11:59:39 CET 2000	ready for commitment (0)
Tue Feb 17 10:15:39 CET 2004	Fri Dec 01 11:59:39 CET 2000	ready for scheduling (1)
Tue Feb 17 10:15:48 CET 2004	Fri Dec 01 11:59:41 CET 2000	ready for reservation (2)
Tue Feb 17 10:15:54 CET 2004	Fri Dec 01 11:59:43 CET 2000	ready for execution (3)
Tue Feb 17 10:15:55 CET 2004	Fri Dec 01 12:29:08 CET 2000	begin execution (4) @ Avatar ID:11
Tue Feb 17 10:15:56 CET 2004	Fri Dec 01 12:35:48 CET 2000	end execution (5) @ Avatar ID:11

Figure 4.18: Simulation GUI in action.

a GUI. A simulation can be built up and executed with the help of a GUI. This GUI offers the availability to define a simulation, run a simulation, and to view the output and the state during a simulation run. Furthermore it is possible to analyze the result after a simulation has been running (see also Section 4.5). Therefore an evaluation facility was added to the simulator to be able to examine the results of a simulation. This offers the opportunity to trace certain properties, for example the CPU load (see Section 4.2.3.1). The evaluation facility can be easily extended if additional properties should be traced.

Although a lot of functions could be added to the simulator there are also functions which could not be realized. Although the NetworkManager is now working properly and all necessary alignments were performed it is not possible to integrate NetworkManagers in the simulation. The integration was not achieved because of different reasons. One important reason is the communication structure of GridSched. Another reason is the design of the simulation which made it impossible to integrate components which are acting both as a server and a client. Examples for components which are acting like that are Titan, DataManager and NetworkManager. In the case of Titan the problem was solved by fully integrating Titan into the simulation. The other components receive requests that can only be answered in a correct way by sending requests to other components. For example, the DataManager needs to question the NetworkManager for a network connection or a NetworkManager might need to ask other NetworkManagers in other domains to produce an answer (see Section 3.8 for details). After sending such a request, the sender waits for an answer to complete the task. The design of the simulation only allows one Avatar to be active at one time. The sender of such an request could not be shut down until an answer is received which could not be produced until the sender was shut down and the receiver could compute an answer.

There are several opportunities to solve this problem. One possible solution would have been to change the components of GridSched in a way that only Titan acts both as a server and a client. This solution was not chosen because it would have been necessary to change the entire design of GridSched and almost every component of the project. Another opportunity is to change the communication structure

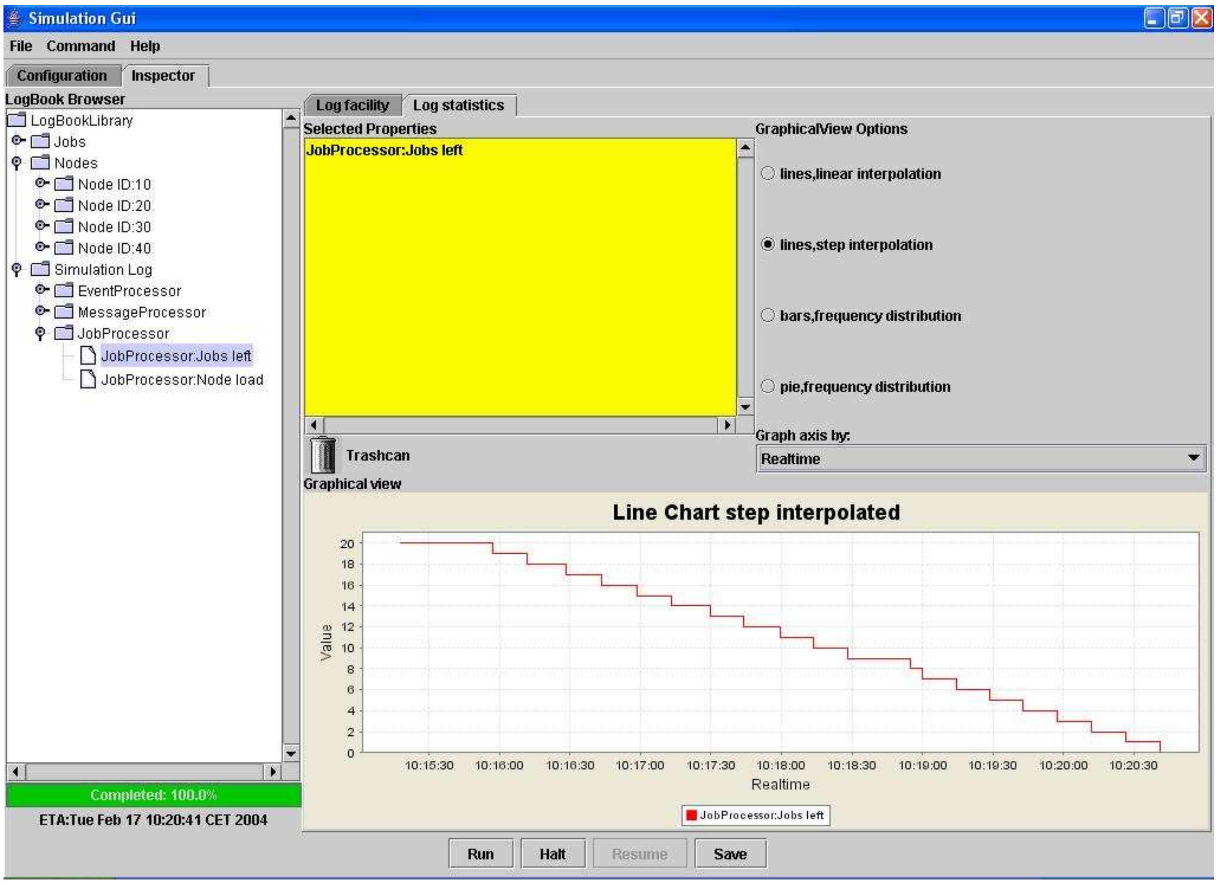
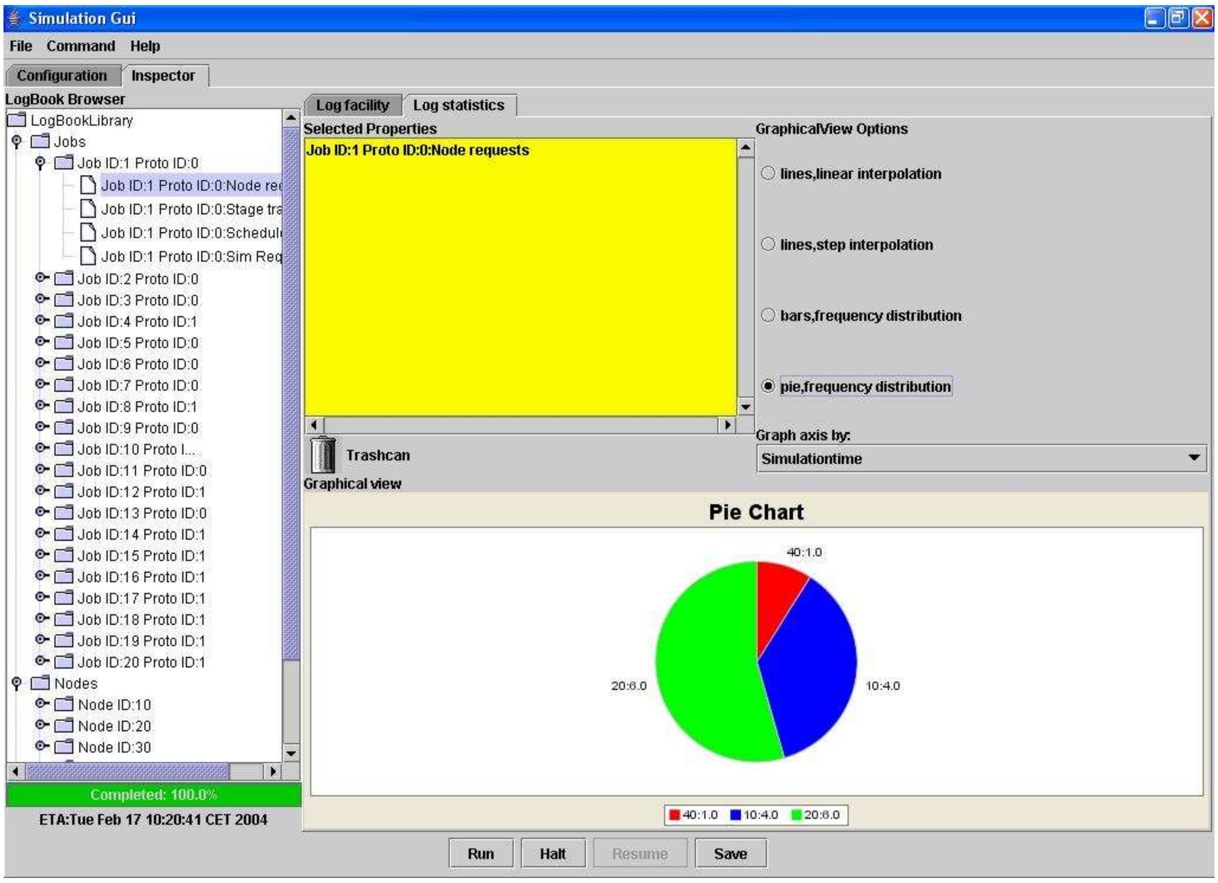


Figure 4.19: Simulation GUI line chart representation of a log entry.

of GridSched to a completely event based system, then components would not have to wait busy for incoming requests or answers and could be shutdown after sending a request. Furthermore it might be possible to allow more than one avatar to be associated to a Node and being activated at once, which would result in a significant design change of the simulator. All of these options would have afforded an amount of work which could not be realized during the remaining time.



LogBookLibrary
- Jobs
  - Job ID:1 Proto ID:0
    - Job ID:1 Proto ID:0:Node req
    - Job ID:1 Proto ID:0:Stage tra
    - Job ID:1 Proto ID:0:Schedul
    - Job ID:1 Proto ID:0:Sim Req
  - Job ID:2 Proto ID:0
  - Job ID:3 Proto ID:0
  - Job ID:4 Proto ID:1
  - Job ID:5 Proto ID:0
  - Job ID:6 Proto ID:0
  - Job ID:7 Proto ID:0
  - Job ID:8 Proto ID:1
  - Job ID:9 Proto ID:0
  - Job ID:10 Proto I...
  - Job ID:11 Proto ID:0
  - Job ID:12 Proto ID:1
  - Job ID:13 Proto ID:0
  - Job ID:14 Proto ID:1
  - Job ID:15 Proto ID:1
  - Job ID:16 Proto ID:1
  - Job ID:17 Proto ID:1
  - Job ID:18 Proto ID:1
  - Job ID:19 Proto ID:1
  - Job ID:20 Proto ID:1
- Nodes
  - Node ID:10
  - Node ID:20
  - Node ID:30

Completed: 100.0%

ETA:Tue Feb 17 10:20:41 CET 2004

Selected Properties: Job ID:1 Proto ID:0:Node requests

GraphicalView Options:
- lines,linear interpolation
- lines,step interpolation
- bars,frequency distribution
- [✓ pie,frequency distribution]

Graph axis by: Simulationtime

🟥 40:1.0 🟦 10:4.0 🟩 20:6.0

Figure 4.20: Simulation GUI pie chart representation of a log entry.

Chapter 5

Summary

5.1 GridSched's Features

GridSched provides the basic functionality to create Jobs on the local computer via a Graphical User Interface and to transfer it to an available Super Scheduler which is able to schedule and reserve the needed resources by querying Local Schedulers, NetworkManagers and DataManagers. If this happened successfully, a Local Scheduler can start and stop the execution of scheduled Jobs. Furthermore the simulation is able to simulate workloads with a huge number of jobs and a Graphical User Interface is able to present the results of the schedule in a flexible manner.

Apart from these functional properties, GridSched provides a flexible and extensible framework for Grid Scheduling. Every single component may either be improved, extended or even be replaced independent of the other components by a more sophisticated one. As communication between the components is done via a communication interface, the underlying protocol may also be replaced. Some ideas for improvements can be found in the next Section. For a comprehensive summary of the features of GridSched see Chapter 3.11.

5.2 Potential improvements and future enhancements

Although considerable effort has been conducted it was not possible to achieve all goals and improvements which emerged since the start of the project. The following list gives a brief overview of planned features and improvements that had to be discarded for various reasons as well as ideas for enhancements. Unless stated otherwise the reason not to implement the feature was the lack of time.

General

- Support of the Globus Toolkit

  At the start of the project Globus Toolkit 2 was soon to become obsolete and therefore disqualified for usage. The usage of Globus Toolkit 3 was prevented due to several reasons. First of all, no final version including all necessary services was available. Furthermore, the complexity of the simulation would have been increased dramatically, as many APIs of the Globus Toolkit would have to be encapsulated. Finally, an additional considerable amount of time would have to be spent on getting familiar with the Globus Toolkit.

Super Scheduler (Titan)

- Interconnection between Super Schedulers

For a better load-sharing it might be helpful to have Super Schedulers communicating to each other as another Super Scheduler may have other Resource Managers available to it. However, whether this measure would result in better performance in reality is questionable.

- Ability to process simultaneous reservations of the same resources

  The current implementation processes the reservations for every Job independently of one another. This may lead to a situation where two scheduled Jobs which include at least two common resources block each other from successfully reserving the required resources as a resource is not available for reservation by another Job as long as it is part of a reservation process.

- Job Monitoring

  A Job Monitor would enable the user to monitor and change the current state of her job.

- Integration of reservation and scheduler

  The integration of the reservation into the scheduler would enable the scheduler to immediately response to a failed reservation by excluding resources during the phase of scheduling.

- A billing system

  This would enable the administrator to keep track of the costs of resource reservation and usage of her users.

DataManager

- No active replica service

  The DataManager in its current implementation does not offer active replica services. This means it does not by itself make decisions about the copying, moving and deletion of data based on recent usage.

- LDAP not used

  Instead of LDAP it was decided to use SQL to store information about files.

NetworkManager

- No Forecast

  The NetworkManager does not offer forecasts of network usage by the network weather system as this would have required an considerable additional amount of time due to increased complexity. Furthermore the possibilities concerning would have been quite limited and due to the fact that the forecasts would have been based on past average usage, the significance of these forecasts are questionable in general.

- No connection to the simulation

  The NetworkManager is not able to communicate with the simulation. A new implementation of the NetworkManager may be required for this to be possible.

- No low-level reservations

  Reservations are not passed down to the level of routers which is of course necessary to enforce reservations.

- No use of GARA

  The Globus Architecture for Advanced Reservation (GARA) is not used as this would have decreased the possibilities of testing the software during the process of implementation.

- Quality of Service

  The realization of this feature would have been quite complex and it would have been quite impossible to test this part of the implementation.

Graphical User Interface

- Graphical User Interface as Java-Applets

  Instead, the Graphical User Interfaces are Java applications using Java Swing.

- No integrated Graphical User Interface

  As the three Interfaces are meant for very different purposes, a integration into one single Graphical User Interface would not result in a significant advantage.

Communication Interface

- No use of SOAP

  For reasons of performance and difficulties during implementation, RMI was preferred.

Language

- Import and Export of XML

  Due to the lack of support for SOAP there was no demand for this feature.

Chapter 6

Configuration and Installation

The software *Apache Ant*is used for automatically installing and configuring GridSched.
Ant is a build tool similar to the UNIX utility *make*. It provides functionality to automate compilation and deployment of the Java-based software in a modular way.
This section will describe the installation process, figure out the different steps and explain how the installation routine can be used.

The GridSched software comprises several components. Depending on the intended use, the configuration of these components may differ. Consider as an example a host that is supposed to offer resources to the Grid but that should not provide scheduling functionality. On this host, the component `Bob` has to be configured correctly, while other components as `Titan` can be deactivated entirely. On the other hand, there are some components (i.e. `Communication` that always have to be installed and configured.

Basically, the installation consists of four steps:

1. Create directory structure
2. Compile Java files
3. Copy binaries to their destination
4. Create initial configuration

The first three steps do not depend on the intended use of the system that is installed. They are required on every system. What can be customized are the paths that should be used by the installation.
The last step is important for determining the functionality of the installed system.

6.1 Ant

This section describes, how Ant is used to meet the requirements explained in the previous paragraph. Before doing this, it should be mentioned that Ant relies on the concept of so-called tasks and targets. Tasks are single actions (i.e compiling or copying a file). These single actions can be aggregated to targets, that can be considered as macros. They can be executed by the user, depending on what should be built by Ant, by passing their name as a command line argument.
Note that there is a default target that is always executed if Ant is called with any parameter. The figure shows which different targets exist for building GridSched and how they depend on each other. Dependencies are an important concept within Ant. Often, several installation steps are interdependent. So for example it is necessary that binaries are created before they can be copied to their intended destination. With Ant, this is modelled by making the target `install` dependent on the target `compile` .
Executing a task leads to Ant executing all the required targets recursively.
All is the default target. It builds GridSched with all options available and does not contain any task

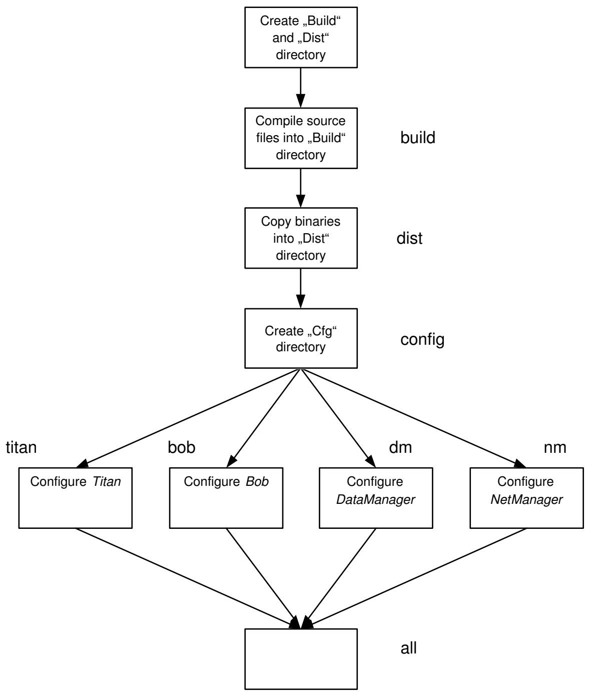


Figure 6.1: Structure of GridSched Ant file.

itself.

```
<project name="GridSched" default="all" basedir=".">
    <description>
        Build file for GridSched
    </description>
  <!-- Global properties -->
  <property name="prop" location="/temp/gridsched2/properties"/>
  <property name="src" location="/temp/gridsched2/src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>
  <property name="cfg"   location="dist/config"/>
  <property name="lib"   location="/temp/gridsched2/lib"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
    <!-- Remove jfree-directory -->
    <!-- delete dir="${src}/org/jfree"/ -->
  </target>

  <target name="communication" depends="init" description="compile communication" >
    <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/communication/" />
        <exclude name="**/test/" />
        <!-- exclude name="**/communication/" / -->
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>

  </target>

  <target name="bob" depends="communication" description="compile bob" >
    <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/bob/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <!-- exclude name="**/bob/" / -->
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="dataManager" depends="bob" description="compile dataManager" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/dataManager/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <!-- exclude name="**/dataManager/" / -->
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="language" depends="dataManager" description="compile language" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/language/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <!-- exclude name="**/language/" / -->
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="netManager" depends="language" description="compile netManager" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/netManager/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <!-- exclude name="**/netManager/" / -->
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>


  <target name="policy" depends="netManager" description="compile policy" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/policy/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <!-- exclude name="**/policy/" / -->
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="scheduler" depends="policy" description="compile scheduler" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
```

```
      destdir="${build}">
    <src path="${src}" />
    <include name="**/scheduler/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <!-- exclude name="**/scheduler/" / -->
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="simulation" depends="scheduler" description="compile simulation" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/simulation/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <!-- exclude name="**/simulation/" / -->
    <exclude name="**/titan/" />
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="titan" depends="simulation" description="compile titan" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/titan/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <!-- exclude name="**/titan/" / -->
    <exclude name="**/util/" />
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>

  <target name="util" depends="titan" description="compile util" >
     <javac sourcepath="${src}" debug="on"
       srcdir="${src}"
       destdir="${build}">
    <src path="${src}" />
    <include name="**/util/" />
        <exclude name="**/test/" />
        <exclude name="**/communication/" />
    <exclude name="**/bob/" />
    <exclude name="**/dataManager/" />
    <exclude name="**/language/" />
    <exclude name="**/netManager/" />
    <exclude name="**/policy/" />
    <exclude name="**/scheduler/" />
    <exclude name="**/simulation/" />
    <exclude name="**/titan/" />
    <!-- exclude name="**/util/" / -->
        <exclude name="**/gui/" />
    <exclude name="**/install/" />
        <classpath>
          <pathelement path="${build}"/>
          <fileset dir="${lib}">
        <include name="**/*.jar"/>
          </fileset>
        </classpath>
     </javac>
  </target>


  <target name="gui" depends="util" description="compile gui" >
     <javac sourcepath="${src}" debug="on"
```

```
      srcdir="${src}"
      destdir="${build}">
   <src path="${src}" />
   <include name="**/gui/" />
       <exclude name="**/test/" />
       <exclude name="**/communication/" />
   <exclude name="**/bob/" />
   <exclude name="**/dataManager/" />
   <exclude name="**/language/" />
   <exclude name="**/netManager/" />
   <exclude name="**/policy/" />
   <exclude name="**/scheduler/" />
   <exclude name="**/simulation/" />
   <exclude name="**/titan/" />
   <exclude name="**/util/" />
       <!-- exclude name="**/gui/" / -->
   <exclude name="**/install/" />
       <classpath>
         <pathelement path="${build}"/>
         <fileset dir="${lib}">
       <include name="**/*.jar"/>
         </fileset>
       </classpath>
    </javac>
</target>


<target name="install" depends="gui" description="compile install" >
    <javac sourcepath="${src}" debug="on"
      srcdir="${src}"
      destdir="${build}">
   <src path="${src}" />
   <include name="**/install/" />
       <exclude name="**/test/" />
       <exclude name="**/communication/" />
   <exclude name="**/bob/" />
   <exclude name="**/dataManager/" />
   <exclude name="**/language/" />
   <exclude name="**/netManager/" />
   <exclude name="**/policy/" />
   <exclude name="**/scheduler/" />
   <exclude name="**/simulation/" />
   <exclude name="**/titan/" />
   <exclude name="**/util/" />
       <exclude name="**/gui/" />
   <!-- exclude name="**/install/" / -->
       <classpath>
         <pathelement path="${build}"/>
         <fileset dir="${lib}">
       <include name="**/*.jar"/>
         </fileset>
       </classpath>
    </javac>
</target>
```

```
<target name="dist" depends="install"
      description="Copy files to their final destination" >
      <mkdir dir="${dist}/class"/>
      <copy todir="${dist}/class">
       <fileset dir="${build}"/>
      </copy>
</target>

<target name="config" depends="dist"
      description="Create initial configuration" >

  <!-- In folder "cfg" information about installed components is stored. Config-GUI uses this in
  <mkdir dir="${cfg}"/>
  <copy todir="${cfg}">
   <fileset dir="${prop}"/>
  </copy>

  <rmic classname="rmic org.gridsched.simulation.elysion.Connector" base="${build}">
      <classpath>
        <pathelement path="${build}"/>
        <fileset dir="${lib}">
      <include name="**/*.jar"/>
        </fileset>
      </classpath>
  </rmic>
  <rmic classname="rmic org.gridsched.communication.RMISession"     base="${build}"/>
</target>


<target name="bob" depends="config"
      description="Create bob configuration" >
  <java classname="org.gridsched.install.Config" classpath="${dist}/class">
          <classpath>
           <pathelement path="${build}"/>
           <fileset dir="${lib}">
            <include name="**/*.jar"/>
           </fileset>
          </classpath>
      <arg line="${cfg}/bobconf.xml bobResource arch value i686"/>
  </java>
</target>

<target name="titan" depends="config"
      description="Create Titan configuration" >
  <java classname="org.gridsched.install.Config" classpath="${dist}/class">
          <classpath>
           <pathelement path="${build}"/>
           <fileset dir="${lib}">
            <include name="**/*.jar"/>
           </fileset>
          </classpath>
      <arg line="${cfg}/titanConf.xml jobBrokerConfig message_timeout value 10000"/>
  </java>
  <java classname="org.gridsched.install.Config" classpath="${dist}/class">
```

```
            <classpath>
             <pathelement path="${build}"/>
             <fileset dir="${lib}">
              <include name="**/*.jar"/>
             </fileset>
            </classpath>
        <arg line="${cfg}/titanConf.xml jobBrokerConfig max_message value 1"/>
    </java>
  </target>


  <target name="dm" depends="config"
        description="Create DataManager configuration" >
    <java classname="org.gridsched.install.Config" classpath="${dist}/class">
            <classpath>
             <pathelement path="${build}"/>
             <fileset dir="${lib}">
              <include name="**/*.jar"/>
             </fileset>
            </classpath>
        <arg line="${cfg}/dmConf.xml database dbhost value localhost"/>
    </java>
  </target>

  <target name="nm" depends="config"
        description="Create NetManager configuration" >
    <java classname="org.gridsched.install.Config" classpath="${dist}/class">
            <classpath>
             <pathelement path="${build}"/>
             <fileset dir="${lib}">
              <include name="**/*.jar"/>
             </fileset>
            </classpath>
        <arg line="${cfg}/nmConf.xml ldap port value 19382"/>
    </java>
  </target>

  <target name="all" depends="bob,titan,nm,dm"
        description="complete installation" >
  </target>




  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

6.2 Graphical User Interfaces

6.2.1 ServerGUI

Before executing the ServerGUI component for the first time the stub and skeleton classes of the `ServiceControl` and the `ConfigFile` class have to be created. This is done by compiling both classes with Java's RMI compiler called RMIC which is included in a Sun JDK like in version 1.4 or better. There is no other parameter than the class path needed when running rmic.

Before compiling the local system's classpath variable has to be modified to refer to the local Java class directory and the JDK directory. As far as the stub and skeleton classes exist and the RMI registry is running in the managed network domain the ServerGUI is ready to be executed by the command `java ServerGUI`.

The default location of the GridSched service's configuration files is the directory containing the `ServerGUI` classes: `.\org\gridsched\gui\server`. If the location is changed the default path may be adjusted in the user interface.

6.2.2 ClientGUI

The GridSched ClientGUI is a graphical user interface for creating and managing jobs on a host running Titan. The client can be started by executing the file `client.jar` or by using the command `java client.jar`.

After starting the program a login window appears (see Figure 6.2).

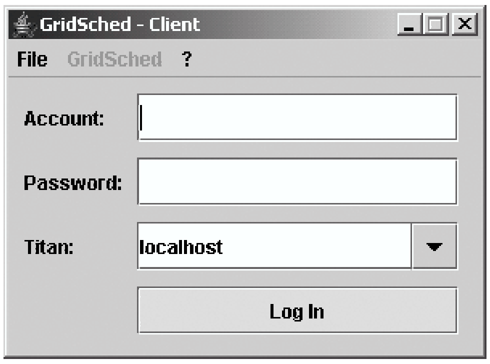


Figure 6.2: The Login window.

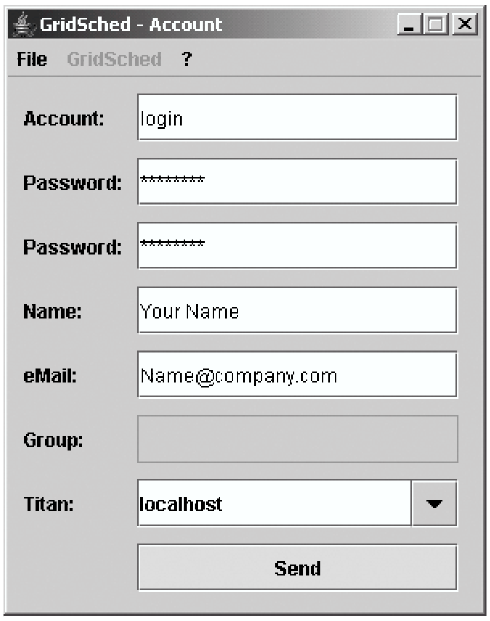
The user has to enter her account name, her password and the address of the Titan she wants to work at. For the login she has to press the "Login" button.

If the user does not have any account, she can create one by opening "File" → "New Account" (see Figure 6.3).

For the creation of the account the user has to enter the complete data and the address of the preferred Titan. After sending the data the "Log In" window appears again.

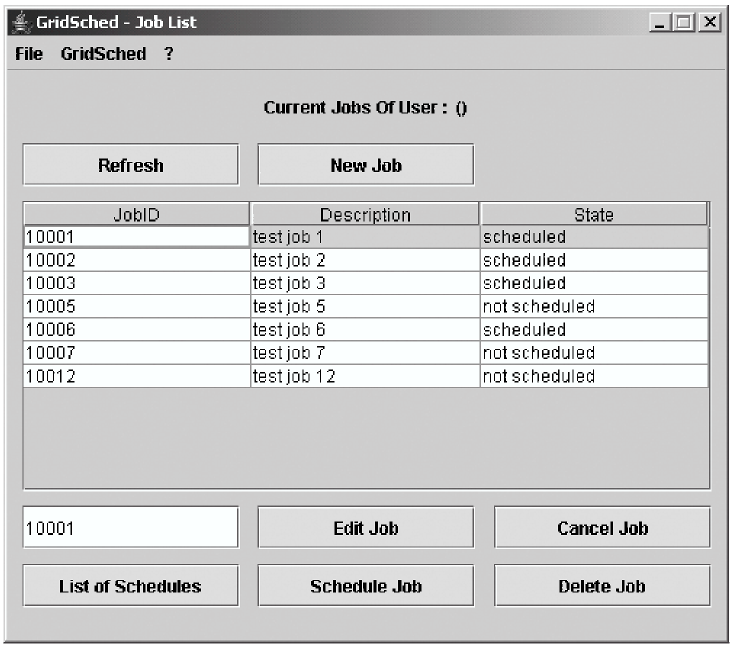
When the user is logged in the "List of Jobs" window appears (see Figure 6.4).

The "List of Jobs" window shows a list of the current jobs. The list contains the "JobID", short descriptions of the jobs and their current schedule state. The user has the possibility to create a new job with the "New Job" button or to work with one of the already existing jobs. She can choose the job she wants to work at by marking it in the list or entering the "JobId" directly. Afterwards she can edit, cancel, delete or start the schedule of the job. If she has edited or cancelled a job she is required to restart the scheduling. If there exists a job which has already been scheduled the user can watch the "List of Schedules" (see Figure 6.5).



Account:	login
Password:	********
Password:	********
Name:	Your Name
eMail:	Name@company.com
Group:	
Titan:	localhost

Figure 6.3: The New Account window.

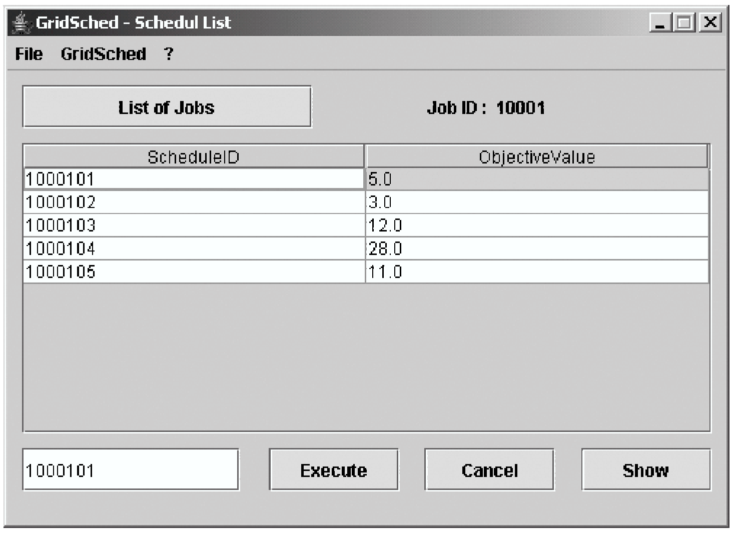


Current Jobs Of User : ()

JobID	Description	State
10001	test job 1	scheduled
10002	test job 2	scheduled
10003	test job 3	scheduled
10005	test job 5	not scheduled
10006	test job 6	scheduled
10007	test job 7	not scheduled
10012	test job 12	not scheduled

10001

Figure 6.4: The List of Jobs window.



Job ID : 10001

ScheduleID	ObjectiveValue
1000101	5.0
1000102	3.0
1000103	12.0
1000104	28.0
1000105	11.0

1000101

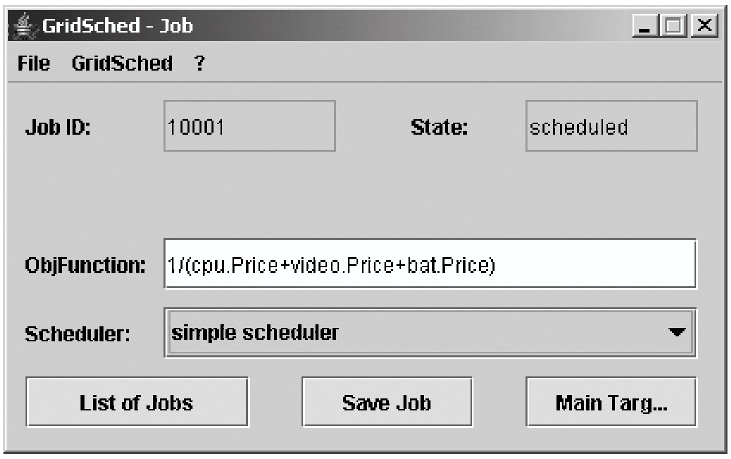
Figure 6.5: The List of Schedules window.

The "List of Schedules" shows all schedules for the selected job with "ScheduleID" and "ObjectiveValue". The user can choose a "ScheduleID" to either execute the job according to the chosen schedule or cancel it. Further details of the schedule are displayed by pressing the "Show" button.

The now opened "Job" window shows all important data of the job: "JobId", "ScheduleState", "ObjectiveFunction" and the scheduler witch created the schedule.

To get back to the "List of Jobs" the "List of Jobs" button is provided (see Figure 6.4).

A new job can be created by pressing the "New Job" button. The "Job" window opens (see Figure 6.6).



Job ID: 10001
State: scheduled
ObjFunction: 1/(cpu.Price+video.Price+bat.Price)
Scheduler: simple scheduler

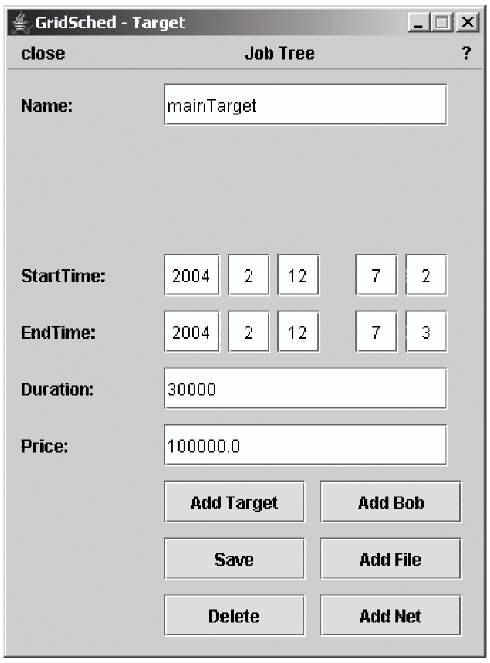
Figure 6.6: The Job window.

To create the job the user has to define the main target by first pressing the "Main Target" button. A "Target" window will be opened (see Figure 6.7).

In the "Target" window the user has to enter a unique name, a start and an end time or an duration and a maximum price for the job. Resources ("Bob", "File", "Net") and sub targets can be added.

In the window "Bob Resource" a hardware resource can be specified. The user is required to enter a name, but all other specifications are optional (see Figure 6.8).

In the window "Data Resource" a file needed can be specified. This requires the name and the hash or address of the file (see Figure 6.9).



Name: mainTarget

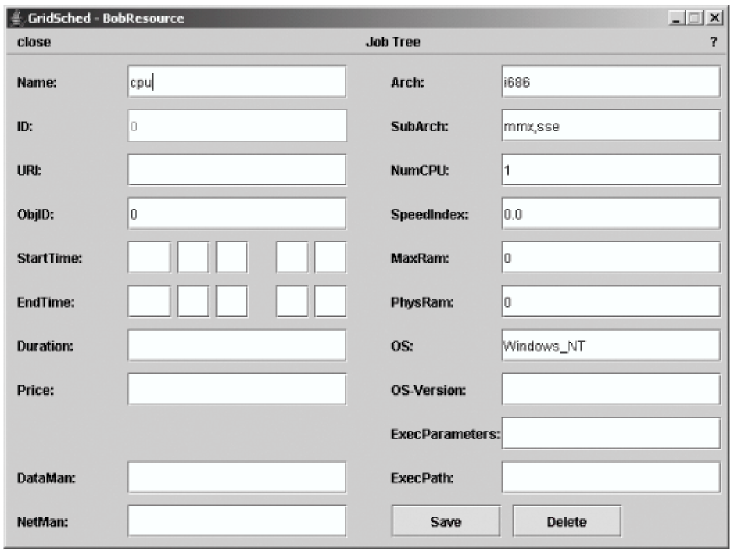
StartTime: 2004 2 12 7 2

EndTime: 2004 2 12 7 3

Duration: 30000

Price: 100000.0

Figure 6.7: The Target window.



Name: cpu

ID: 0

URI:

ObjID: 0

StartTime:

EndTime:

Duration:

Price:

DataMan:

NetMan:

Arch: i686

SubArch: mmx,sse

NumCPU: 1

SpeedIndex: 0.0

MaxRam: 0

PhysRam: 0

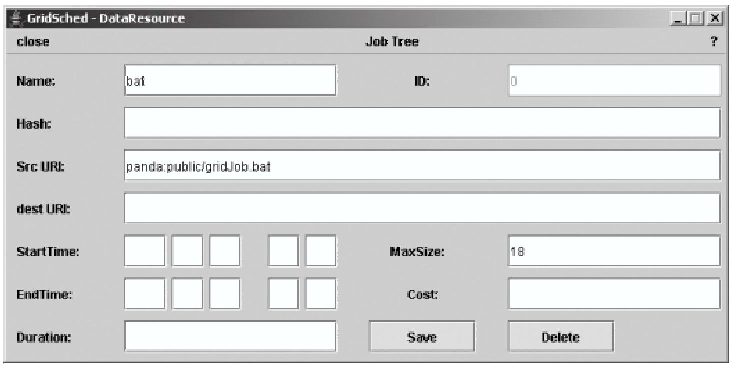
OS: Windows_NT

OS-Version:

ExecParameters:

ExecPath:

Figure 6.8: The Bob Resource window.



Name: bat

ID: 0

Hash:

Src URI: panda:public/gridJob.bat

dest URI:

StartTime:

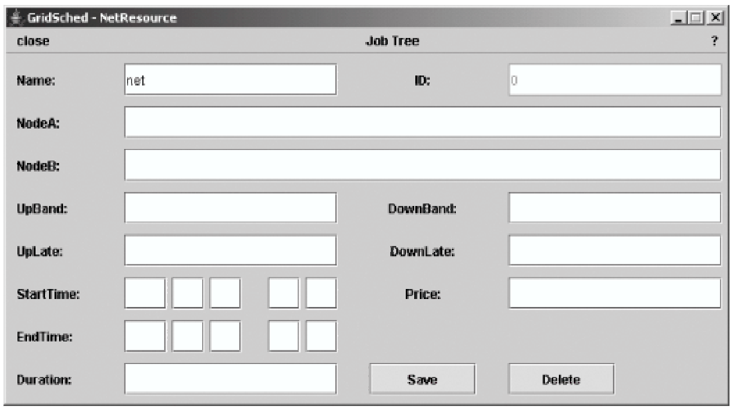
EndTime:

Duration:

MaxSize: 18

Cost:

Figure 6.9: The Data Resource window.



Name:	net	ID:	0
NodeA:			
NodeB:			
UpBand:		DownBand:	
UpLate:		DownLate:	
StartTime:		Price:	
EndTime:			
Duration:			

Figure 6.10: The Net Resource window.

In the window ”Net Resource” a network connection can be specified. The name is mandatory, all other fields are facultative (see Figure 6.10).

6.3 Scheduler

6.3.1 Resource Information Service

GridSched uses Sun ONE Directory Server [18] as an LDAP Server. The ”properties” folder contains the appropriate configuration ldif file (`99user.ldif`). It may be necessary to modify that file in order to use it on other LDAP implementations.

6.3.2 Titan

Most of the Super Scheduler’s properties may be configured by a central XML file which is situated in GridSched’s ”properties” folder. It is named `titanConf.xml`

```
<jobBrokerConfig>
    <message_timeout value = "10000"/>
    <max_messages value = "1"/>
</jobBrokerConfig>
```

Configure how long Titan will wait for replies when communicating with other GridSched components and how many replies Titan will accept for the same request.

```
<reservationClientConfig>
    <message_timeout value = "5000"/>
</reservationClientConfig>
```

Configure how long Titan will wait for replies to reservation requests.

```
<JNDIConfig>
    <host name = "panda.e-technik.uni-dortmund.de" port = "19382"/>
</JNDIConfig>
```

Configure the hostname and port that Titan will use to contact the remote directory service.

```
<nuSchedConfig>
    <optimizer_timeout value = "30000"/>
    <solver_timeout value = "15000"/>
    <solver_max_fail value = "2"/>
    <solver_max_mutate value = "4"/>
</nuSchedConfig>
```

Configure properties of the NuSched scheduling module. Define the maximum time that the solver may spend on creating one schedule, how often one scheduling-task may fail before it is abandoned and how often it may be mutated. Specify how long the total scheduling process is allowed to take.

```
<agentConfig>
    <request_timeout value = "30000"/>
    <negative_cache_timeout value = "30000"/>
    <positive_cache_timeout value = "30000"/>
    <depth_first_mode value = "false"/>
</agentConfig>
```

Configure properties of NuSched’s agent modules. Specify how long requests are allowed to remain inside an agent and how long entries will be cached. Agent components may be in depth first mode and evaluate one option after another, or they may be in breath first mode and try to evaluate all options in parallel.

```
<defaultConfig>
    <debugUI value = "true"/>
</defaultConfig>
```

Choose whether or not a debug window should be displayed to to visualize some of the activities going on in Titan.

```
<Policies>
    <Policy group="*" user="*" name="access">
        <permission general = "true"/>
    </Policy>
</Policies>
```

Set up basic access permissions for Titan.

Most properties will be stored in Titan's "properties" Hashtable. Titan features a static method `Object getProperty(String name)` to retrieve any property from anywhere in the code at any time.

6.3.3 Bob

Most of the LocalScheduler's properties can be configured by the XML file `bobconf.xml` in GridSched's "properties" folder.

```
<JNDIConfig>
    <host name = "panda.e-technik.uni-dortmund.de" port = "19382"/>
</JNDIConfig>
```

Define the hostname and port that Bob will use to communicate with the remote directory service, where it publishes its resources.

```
<ReservationConfig>
    <grace_timeout value= "10000"/>
</ReservationConfig>
```

Configure the duration that a grace reservation will be valid.

```
<bobResource>
        <arch value = "i686"></arch>
        <subArch value ="mmx,sse"></subArch>
        <os value = "Windows_NT"></os>
        <osVer value = "5.11"></osVer>
        <speedIndex value="2400"/>
        <physRam value="502"/>
        <maxRam value="1266"/>
        <id value="814"/>
        <OID value="123456"/>
        <name value="thisMachine"/>
        <numCPU value="1"/>
        <!-- uri value="grid://pool-e08/bob/thisMachine"/ -->
        <!-- bob value="grid://pool-e08/bob"/ -->
        <!-- dataManager value="grid://pool-e08/data"/ -->
        <netManager value="grid://pool-e06/net"/>
        <diskSpace value="9767516"/>
        <cancelPrice value="1"/>
        <bindingPrice value="1"/>
        <gracePrice value="1"/>
        <type value="0"/>
        <cost value="0"/>
</bobResource>
```

Set up the properties of the resource(s) that this Bob manages. There may be more than one.

```
<Policies>
    <Policy group="*" user="*">
        <permission general="true"/>
        <costfactor value="0.001"/>
    </Policy>
</Policies>
```

Set up basic access permissions for Bob and its' resources.

Unless not specified by the configuration file Bob will try to determine the following resource properties by itself:

- `id` Bob will compute a unique id from the machine's IP-address.
- `uri` Bob will use the uri `grid:\\"hostname"\bob\"resourceName"`.
- `bob` Bob will use the uri `grid:\\"hostname"\bob`.
- `dataManager` Bob will use the uri `grid:\\"hostname"\data`.
- `netManager` Bob will use the uri `grid:\\"hostname"\"name"`.

6.4 Data Manager

Four different aspects of configuration are addressed by the configuration.

- Network Manager
- Database
- Scheduler
- gassGate

The first aspect is configured by keys located directly beneath the file's ¡configuration¿ element. For the other topics, separate sections have been created.
The section **gassGate** contains the information necessary for accessing a Globus GASS Gateway.
The keys withing **scheduler** allow to configure the interaction between DataManager and Scheduler.
The section **database** contains the information necessary for accessing the database used by `DataManager`.

6.5 Network Manager

The configuration file consists of two section. The first one, **localGraph**, contains the key **location**. This is used to specify the location of the file which describes the network graph.
The second section, **ldap**, contains the keys **address** and **value**. They are used to specify IP address and TCP port of the LDAP server used by `NetManager`.

```
<configuration>
        <netManagerHostName value="pool-b07"/>
        <netManagerIPAddress value="129.217.186.163"/>
        <database>
                <dbHost value ="panda"/>
                <dbName value ="gridsched"/>
                <dbTable value ="file"/>
                <dbUser value ="pg428"/>
                <dbPasswd value ="aaa"/>
                <providedSpace value = "10000000"/>
        </database>

        <scheduler>
                <startDelay value ="1000"/>
                <validityDuration value ="120000"/>
                <localHost value ="129.217.184.3"/>
                <localFolder value="c:/temp"/>
        </scheduler>

        <gassGate>
                <port value ="21"/>
                <ftpUser value ="griduser"/>
                <ftpPasswd value ="grid"/>
        </gassGate>

</configuration>
```

Figure 6.11: Sample configuration file for `DataManager`.

```
<configuration>
        <localGraph>
                <location value="c:\home\LocalGraph.ser"/>
        </localGraph>
        <ldap>
                <address value="localhost"/>
                <port value="19382"/>
        </ldap>
</configuration>
```

Figure 6.12: Sample configuration file for `NetManager`.

6.6 Simulator

This section describes all necessary steps that are needed to create a simulation scene. In contrast to other parts of GridSched, there is no configuration in general. All program settings are individual for each scene, and are parts of the simulation´s input. As mentioned in 4.5, there are two possibilities to committed an input to the simulation. The first one is provided by a graphical user-interface introduced in 4.5. The latter one is the use of an XML-document, which contains all settings. Section gives a brief introduction how to create these. But both require a remote directory and a SQL-database server to manage resources, refer to Section 6.4 for instructions.

6.6.1 Creating a simulation scene by an XML-Document

The simulation GUI provides an option to load an XML-file, which was either created externally by hand or via the internal text editor (see 4.5). The syntax equals to common XML-tag definitions:

1. `empty tags` $< defiervalue = "value\_of\_defier" / >$
2. `enclosing tags` $< defier > ... < /defier >$ These tags may include `empty tags`.

The document is structured in three parts, global simulation settings (see 6.6.1.1), a list of jobs that are to be simulated (see 6.6.1.2), the virtual Grid Computing Environment (6.6.1.3). All tags of any part of the document has to be encapsulated with the enclosing tag '`<simulation> ... </simulation>`'. The text below introduces in each part in more detail.

6.6.1.1 Global Simulation Settings

These settings affect the whole configuration of the simulator and the simulation turn. The list below gives an overview of each tag defier which must be set.

- `name` The name of the simulation.
- `directory` The working-path of the simulation (a an absolute canonical path name).
- `staticResourceLibrary` A Remote directory server for the simulation to store resources, its network-address (`address`) and port number (`port`) must be set.
- `starttime - endtime` A start time and end time of the simulation (GST).
- `rmiport` The IP-port that should be used for RMI communication.
- `logunit` The name for a workload file (without suffix as a canonical path name).
- `SQLdatabase` The configuration for the SQL database used for the DataManager component, which consists of following settings: host-address (`dbHost`) of the SQL-server, the name of the database (`dbName`), the table (`dbTable`), the table user (`dbUser`) and her password (`dbPasswd`).

Their sequence in the document is not prescribed an can vary. This block is encapsulated by the `<configuration>...</configuration>` tag. An example configuration of this part of the document is given below:

```
<configuration>
                <directory value="c:\temp\simulation\"/>
    <name value="foobar"/>
    <starttime value="1.12.2000 8:00:00"/>
    <endtime value="31.12.2010 8:00:00"/>
    <rmiport value="2099"/>
    <staticResourceLibrary>
        <address value="foo.bar.de"/>
        <port value="19381"/>
    </staticResourceLibrary>
    <logUnit>
        <filename value="c:\temp\simulation\logfile"/>
    </logUnit>
    <SQLdatabase>
    <dbHost value ="foo.bar.de"/>
    <dbName value ="simulation"/>
    <dbTable value ="turn_01"/>
    <dbUser value ="foobar"/>
    <dbPasswd value ="foobar"/>
    </SQLdatabase>
</configuration>
```

6.6.1.2 Job Definitions

The XML-document also contains the description of jobs which should be simulated. A job consists of several different parts. The most important setting is the simulation time of the submission to the scheduler, because it affects the sequence of the simulation (`simSchedTime`). On top of this the user who submits the job and the group the user belongs to must be set, to comply the policies used by Bob and Titan. All other tags describing a job are equal to the familiar definitions as used in reality (see 3.3). Their is no limitations, how many jobs are committed to the document except of hardware limitations given by the hosted machine. Below an example is given, how to specify a single job:

```
<job>
        <simSchedTime value="01.12.2000 12:00:00"/>
        <user value="leading0"/>
        <group value="pg428"/>
        <objectiveFunction  value = "1/(cpu.Price+archive.Price+bat.Price)"/>
        <objectTarget>
            <doItem>
                <instructions value="run(bat,cpu,archive)"/>
            </doItem>
            <resources>
                <bobResource>
                    <arch value = "athlonxp"/>
                    <subArch value ="mmx,sse"/>
                    <os value = "Windows_NT"/>
                    <name value = "cpu"/>
                    <numCPU value = "1"/>
                </bobResource>
                <dataResource>
                    <name value = "bat"/>
                    <maxSize value = "18"/>
                    <sourceURI value = "grid://41/public/extractproject.bat"/>
                </dataResource>
                <dataResource>
                    <name value = "archive"/>
                    <hash value = "project"/>
                </dataResource>
            </resources>
            <name value="main"/>
            <oid value="10"/>
            <price>
                <value value="7400000"/>
                <currency value="0"/>
            </price>
            <timeslot>
                <starttime value="3.12.2000 20:00:00"/>
                <endtime value="5.12.2000 20:30:00"/>
                <duration value="7200000"/>
            </timeslot>
        </objectTarget>
    </job>
```

To be able to simulate a big amount of jobs it would be uncomfortable to enter all these jobs into the XML-document by hand. Therefore it is possible to use the described job definitions as templates (`protos`) for creating jobs of the same type, varied by the submission time (`simSchedTime`), the `starttime` and `endtime` and the `duration`. To invoke the production of jobs from a single template, the '`<workload value="workload_value"/>'''` must be included to the document at job-level. To vary the mentioned values, a randomized method is used. To gain more control over it , the `workload_value` contains a tuple of ten long values of different assignments parted by '`|`' characters. These values has the following meaning and must appear the the same ordering:

1. **JobProtoType** The ID of the job template that should be used, the ID numbers are given by the sequence of the job templates in the documents, starting with '0'.
2. **Count** The number of jobs to produce from the referred template.
3. **SubMissionIntervall** The submission-time gap between two jobs as a long value in milliseconds.
4. **SubmissionVariance** The randomized variance of the `SubMissionIntervall` in milliseconds.
5. **PrefferedStartTimeOffset** The time gap between submission time and `starttime` of a job in milliseconds.
6. **StartTimeOffsetVariance** The randomized variance of the `PrefferedStartTimeOffset` in milliseconds.
7. **PrefferedEndTimeOffset** The time gap between the begin of the `starttime` and `endtime` of a job in milliseconds.
8. **EndTimeVariance** The randomized variance of the `PrefferedEndTimeOffset` in milliseconds.
9. **PrefferedDuration** The `duration` of a job.
10. **DurationVariance** The randomized variance of the `PrefferedDuration` in milliseconds.

Note that the submission-time of the first job is relative to the `starttime` of the simulation and not of its used template. A restriction of how many workload tags are defined is not given. It is even possible to use a template more than once. An example is given below:

```
<workload value="0|10|14400000|36000000|1800000|360000|7200000|36000|400000|0"/>
```

6.6.1.3 Grid Environment

The XML-document must also contain all required information to create a Grid Computing Environment for the simulation, which consists of Nodes. As described in 4.1, a Node represents a computer on which services of GridSched like Bob and DataManager can run. Nodes are defined in the document with the enclosing tag '`<node>...</node>`' which must contain following fields

- **description** A string to describe this Node.
- **id** The virtual network address for this node. Note, that this number must divisible by 10, refer to 4.2.2.1 for details.
- **ipAdress** A real IPv4 address can be attached, but this has no effect on the simulator since virtual network addresses are used.
- **avatar** Is an enclosing tag, to attach a new Avatar to the Node, and contains the fields, `type`, `aliasLocation`, `aliasParameter`. The first one indicates the service, that it should be an instance of, the digit '1' is to be set for a Bob, '2' for a DataManager. The second one gives the executable, that is to be called, when the associated Node is scheduled to run. The last one represents optional parameter, that can be passed to the executable. A node cannot carry more than one Avatar in the current implementation of Elysion, but the data-structures and the initialization of the Simulation can handle multiple Avatars for a Node. An example configuration of a Node is given below:

```
<node>
        <description value="Computer Node 10"/>
        <id value="10"/>
        <ipAddress value="129.186.217.10"/>
        <avatar>
            <type value="1"/>
            <aliasLocation value="java org.gridsched.bob.Bob"/>
            <aliasParameter value=""/>
            </avatar>
</node>
```

At least to complete a simulation scene, there must be resources be defined. Otherwise every schedule from Titan will fail. There are two types of resources, for Bob and for the DataManager. Both are represented by the enclosing tags `<bobResource>...</bobResource>` and `<databaseFile>...</databaseFile>`. The empty tags to fill in are equal to their Java class´representations and are explained in 3.3.3.4 and 3.7.3.3. To attach a resource to services, that have to manage it, the following fields must be set to their URIs where their virtual address is encoded:

- For Bob Resources, the `URI`, `bob`, `dataManager` and `netManager` fields. An example is given below for a Bob with the virtual network address '11', a DataManager addressed by '22'. The `netManager` has no effect and will be ignored by the simulation´s initialization.

  ```
  <URI value="grid://11"/>
  <bob value="grid://11/bob/thisMachine"/>
  <dataManager value="grid://22/data"/>
  <netManager value="grid://33/net"/>
  ```

- For DataManager Resources, the `uri` and `sourceURI` fields have to be set to dedicated locations as an URI formatted string. Note that real data transfers of the DataManager are not simulated by the simulation, so that these setting do not affect. But the scheduling and reservation of these are simulated, so that declarations of Data Resources are mandatory. An example of a Data Resource is given below:

  ```
  <databaseFile>
      <hash value="astronomical"/>
      <description value="programming project files"/>
      <author value="123"/>
      <state value="0"/>
      <uri value="panda://public/image.gz"/>
      <SourceURI value="panda://public/image.gz"/>
      <mark value="1"/>
      <dateFrom value="1.12.1998 5:00:00"/>
      <dateTo value="12.12.2010 8:00:00"/>
      <filesize value="123"/>
      <filedate value="1.12.1999 8:00:00"/>
      <cost value="50"/>
  </databaseFile>
  ```

The XML-document is parsed during the initialization of the simulation environment. Within this process the Java objects are produced and for example given to Okeanos, the internal database of the simulation to store these definitions. Another example are the Bob Resources which are instantiated corresponding to the values in the file and then are pushed to the StaticResourceLibrary which writes the information into an LDAP server.

At least an example of a simulation XML-document is given below. It defines a scene with two job templates, four nodes and 4 resources. For each template, ten jobs are generated resulting in twenty jobs to be simulated.

Example XML document for a simulation scene:

```
<simulation>
<configuration>
<directory value="c:\temp\gridsched2\simulation\"/>
<name value="SimulationsTest"/>
<starttime value="1.12.2000 8:00:00"/>
<endtime value="31.12.2010 8:00:00"/>
<rmiport value="2099"/>
<staticResourceLibrary>
<address value="panda"/>
<port value="19381"/>
</staticResourceLibrary>
<logUnit>
<filename value="c:\temp\gridsched2\simulation\presentationlog"/>
</logUnit>
<SQLdatabase>
<dbHost value ="panda"/>
<dbName value ="gridsched"/>
<dbTable value ="gridsched"/>
<dbUser value ="pg428"/>
<dbPasswd value ="aaa"/>
</SQLdatabase>
</configuration>

<job>
<simSchedTime value="01.12.2000 12:00:00"/>
<user value="leading0"/>
<group value="pg428"/>

<objectiveFunction  value = "1/(cpu.Price+archive.Price+bat.Price)"/>

<objectTarget>

<doItem>
<instructions value="run(bat,cpu,archive)"/>
</doItem>

<resources>
<bobResource>
<arch value = "athlonxp"/>
<subArch value ="mmx,sse"/>
<os value = "Windows_NT"/>
<name value = "cpu"/>
<numCPU value = "1"/>
</bobResource>
<dataResource>
<name value = "bat"/>
<maxSize value = "18"/>
<sourceURI value = "grid://41/public/extractproject.bat"/>
</dataResource>
<dataResource>
<name value = "archive"/>
<hash value = "project"/>
</dataResource>

</resources>
```

```
<name value="main"/>
<oid value="10"/>

<price>
<value value="7400000"/>
<currency value="0"/>
</price>

<timeslot>
<starttime value="3.12.2000 20:00:00"/>
<endtime value="5.12.2000 20:30:00"/>
<duration value="7200000"/>
</timeslot>

</objectTarget>
</job>


<job>
<simSchedTime value="01.12.2000 14:00:00"/>
<user value="leading0"/>
<group value="pg428"/>

<objectiveFunction  value = "1/(cpu.Price+astro.Price+bat.Price)"/>

<objectTarget>

<doItem>
<instructions value="run(bat,cpu,astro)"/>
</doItem>

<resources>
<bobResource>
<arch value = "athlonxp"/>
<subArch value ="mmx,sse"/>
<os value = "Windows_NT"/>
<name value = "cpu"/>
<numCPU value = "1"/>
</bobResource>
<dataResource>
<name value = "bat"/>
<maxSize value = "18"/>
<sourceURI value = "grid://41/public/astro.bat"/>
</dataResource>
<dataResource>
<name value = "astro"/>
<hash value = "project"/>
</dataResource>

</resources>

<name value="main"/>
<oid value="10"/>

<price>
<value value="7400000"/>
```

```
<currency value="0"/>
</price>

<timeslot>
<starttime value="3.12.2000 20:00:00"/>
<endtime value="5.12.2000 20:30:00"/>
<duration value="7200000"/>
</timeslot>

</objectTarget>
</job>


<node>
<description value="Computer Node 40"/>
<id value="40"/>
<ipAddress value="129.186.217.40"/>
<avatar>
<type value="1"/>
<aliasLocation value="java org.gridsched.bob.Bob"/>
<aliasParameter value=""/>
</avatar>

</node>


<node>
<description value="DataManager Node 30"/>
<id value="30"/>
<ipAddress value="129.186.217.30"/>
<avatar>
<type value="2"/>
<aliasLocation value="java org.gridsched.dataManager.DataGate"/>
<aliasParameter value=""/>
  </avatar>

</node>


<node>
<description value="DataManager Node 20"/>
<id value="20"/>
<ipAddress value="129.186.217.20"/>
<avatar>
<type value="2"/>
<aliasLocation value="java org.gridsched.dataManager.DataGate"/>
<aliasParameter value=""/>
</avatar>

</node>


<node>
<description value="Computer Node 10"/>
<id value="10"/>
<ipAddress value="129.186.217.10"/>
```

```
<avatar>
<type value="1"/>
<aliasLocation value="java org.gridsched.bob.Bob"/>
<aliasParameter value=""/>
</avatar>

</node>

<bobResource>
<arch value = "athlonxp"></arch>
<subArch value ="mmx,sse"></subArch>
<os value = "Windows_NT"></os>
<osVer value = "5.11"></osVer>
<speedIndex value="2400.0"/>
<physRam value="502"/>
<maxRam value="1266"/>
<id value="0"/>
<OID value="123456"/>
<type value="0"/>
<cost value="1"/>
<name value="thisMachine"/>
<numCPU value="1"/>
<URI value="grid://11"/>
<bob value="grid://11/bob/thisMachine"/>
<dataManager value="grid://22/data"/>
<netManager value="grid://33/net"/>

</bobResource>
<bobResource>
<arch value = "athlonxp"></arch>
<subArch value ="mmx,sse"></subArch>
<os value = "Windows_NT"></os>
<osVer value = "5.11"></osVer>
<speedIndex value="2400.0"/>
<physRam value="502"/>
<maxRam value="1266"/>
<id value="0"/>
<OID value="123456"/>
<type value="0"/>
<cost value="1"/>
<name value="thisMachine"/>
<numCPU value="1"/>
<URI value="grid://41"/>
<bob value="grid://41/bob/thisMachine"/>
<dataManager value="grid://22/data"/>
<netManager value="grid://33/net"/>

</bobResource>



<databaseFile>

<hash value="astronomical"/>
<description value="programming project files"/>
<author value="123"/>
```

```
<state value="0"/>
<uri value="panda://public/image.gz"/>
<SourceURI value="panda://public/image.gz"/>
<mark value="1"/>
<dateFrom value="1.12.1998 5:00:00"/>
<dateTo value="12.12.2010 8:00:00"/>
<filesize value="123"/>
<filedate value="1.12.1999 8:00:00"/>
<cost value="50"/>

</databaseFile>


<databaseFile>

<hash value="project"/>
<description value="hubble space scope prints"/>
<author value="123"/>
<state value="0"/>
<uri value="panda://public/astro.gz"/>
<SourceURI value="panda://public/astro.gz"/>
<mark value="1"/>
<dateFrom value="1.12.1998 5:00:00"/>
<dateTo value="12.12.2010 8:00:00"/>
<filesize value="123"/>
<filedate value="1.12.1999 8:00:00"/>
<cost value="50"/>
</databaseFile>


<!-- #
# Elysion Workload Definition File
# Format for each Line representing a prototype population:
# JobProtype(int)|Count(int)|SubmissionInterval(long)|
# SubmissionVariance(long)|PrefferedStartTimeOffset(long)|
# StartTimeOffsetVariance(long)|prefferedEndTimeOffset(long)|
# EndTimeVariance(long)|
# PrefferedDuration(long)|DurationVariance(long)
#
-->

<workload value="0|10|14400000|36000000|1800000|360000|7200000|36000|400000|0"/>
<workload value="1|10|14400000|36000000|1800000|360000|7200000|36000|400000|0"/>
</simulation>
```

List of Figures

Bibliography

[1] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, Paul F. Reynolds Jr. A Synopsis of the Legion Project. Department of Computer Science, University of Virginia.

[2] Arcot K. Rajasekar, Reagan W. Moore. Data and Metadata Collections for Scientific Applications. In *Lecture Notes in Computer Science Vol. 2110, Springer-Verlag Heidelberg*, 2001.

[3] B. Lakhal, S. Pinschke. Netzwerk-Management. In *Seminarband PG 428: Entwicklung eines Daten- und Netzwerk-Schedulings für Grid-Computing in Verbindung mit einem flexiblen Bewertungsmechanismus*, 2003.

[4] *The Condor Project, http://www.cs.wisc.edu/condor/, as of March 2004.*

[5] Dietmar Erwin. UNICORE Plus Final Report, Uniform Interface to Computing Resources, Joint Project Report for the BMBF Project UNCORE Plus. Forschungszentrum Jülich, 2003.

[6] *The EU Datagrid, http://www.eu-datagrid.org, as of March 2004.*

[7] *Global Grid Forum, http://www.gridforum.org, as of March 2004.*

[8] *Globus Alliance, http://www.globus.org, as of March 2004.*

[9] Heinz Stockinger, Asad Samar, Bill Allcock, Ian Foster, Koen Holtmann, Brian Tierney. File and Object Replication in Data Grids. In *Proc. 10th Intl. Symp. on High Perfomance Distributed Computing, IEEE Press*, 2001.

[10] J. Bester, I. Foster, C. Kesselmann, J. Tedesco, S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.

[11] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, California, August 7-9 2001.

[12] *http://java.sun.com.*

[13] Kay S. Brennecke, Stefan Einbrodt, Jan Philip Eumann, Sebastian Freitag, Joern Gerendt, Manuel Heß, Bouchta Lakhal, Stefan Pinschke, Zouhair Sabry, Daniel Sander, Sebastian Schlitte, Thomas Wojczechowski. Interim report pg 428: Development of a data- and network scheduling for grid-computing including a flexible evaluation mechanism. Technical report, University of Dortmund, 2003.

[14] *Legion, http://legion.virginia.edu, as of March 2004.*

[15] OpenLDAP. Openldap - community developed ldap software. 2004.

[16] S. Pinschke, S. Schlitte. Directory-Dienste mit LDAP. In *Seminarband PG 428: Entwicklung eines Daten- und Netzwerk-Schedulings für Grid-Computing in Verbindung mit einem flexiblen Bewertungsmechanismus*, 2003.

[17] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. In *Global Grid Forum Draft Recommendation, 6/27/2003.*

[18] *http://wwws.sun.com/software/products/directory_srvr/home_directory.html.*

[19] Thomas Sandholm, Jarek Gawor. Globus Toolkit 3 Core - A Grid Service Container Framework. July 2003.

[20] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke. Data Management and Transfer in HighPerformance Computational Grid Environments. In *Parallel Computing, Vol. 28*, 2001.

[21] W. Allcock, J. Bresnahan, I. Foster, L. Liming, J. Link, P. Plaszczac. GridFTP Update January 2002. In *Technical Report (http://www.globus.org/datagrid/gridftp.html)*, 2002.

[22] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, et al. Data Management in an International Data Grid Project. In *Lecture Notes in Computer Science Vol. 1971, Springer-Verlag Heidelberg*, 2000.