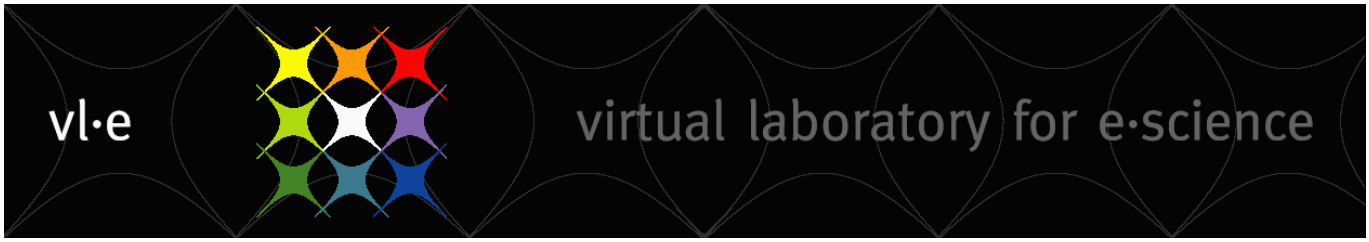

Virtual Laboratory for e-Science
Grid enabled Virtual Laboratory Amsterdam

The WS-VLAM Users' Guide

(Version 0.1)

gvlam@lists.vl-e.nl

Informatics Institute
Universiteit van Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Information:
Mail: gvlam@lists.vl-e.nl
Web: http://www.vl-e.nl

Contents

Chapter 1

Introduction

This is a prototype release of the WS-VLAM software.

The WS-VLAM environmnet provides a science portal for distributed experimentation in applied scientific research. It offers scientists remote experiment control, data management facilities and access to distributed resources by providing cross-institutional integration of information and resources in a familiar environment.

The WS-VLAM environment is currently deployed on the rapid prototyping environment (DAS3 cluster of UvA). In the near Future (autumn 2007), the WS-VLAM software will be also deployed on the more stable Proof-of-Concept environment maintained by SARA (http:www.sara.nl/userinfo/grid/description). However, in principle you can also use the WS-VLAM environmnet on any Grid (Globus) enabled system.

This step-by-step document describes the procedure needed by the WS-VLAM end-user to use the WS-VLAM environment to run their experiments and to create your own modules which you can use in your experiments.

At the initial stage of the VL-e project, the WS-VLAM environmnet provides a minimal set of features needed to start experimenting with the basic concepts introduced by the VL-e project. The WS-VLAM environmnet allows among other things to:

- Compose and execute application workflow dataflow.
- Provides interfaces to develop and port existing code written in C/C++, JAVA, and python.
- Provides a wrapper for legacy application to be used as workflow components
- Provides an tool to convert Web services into worklfow components
- Provides tools to define, annotate, search for workflow component
- For long running application workflow, user does not need to be loged all the time to monitor the execution of the workflow.
- Provides a monitoring facilities based on the WS-notification standard implemented by the Grid Middleware.

In the next phases of the VL-e project, new generic features and enhancements will be added to the WS-VLAM environmnet, based on the feedback and feature requests provided by the users of WS-VLAM environmnet.

This document introduces you to the WS-VLAM environment and is divided into the following parts:

- Part I: Introduction to its concepts and the WS-VLAM experiment model (chapter 2).
- Part II: Installing and configuring your WS-VLAM environment (chapter 3).
- Part III: Using the Frontend (chapter 4) to edit, set up and run your workflow.
- Part IV: Creating your own modules (chapter 5) which you can use in your own experiments.

Please report any problems, bugs to: `gvlam@lists.vl-e.nl` with the subject: '`bug report`'.

Chapter 2

The WS-VLAM workflow Model

2.1 The Vision

The aim of the WS-VLAM system is to provide and support coordinated execution of distributed Grid-enabled components combined in a workflow. This system combines the ability to take advantage of the underlying Grid infrastructure and a flexible high-level rapid prototyping environment. On the high level a distributed application is composed as a data driven workflow where each component represents a process or service on the Grid. Processes are activated only when the data is available on their input ports. The significant difference from other similar systems is the support for simultaneous execution of co-allocated processes on the Grid which enables direct data streaming between the distributed components: traditional batch processing of grid jobs and workflow execution based on input/output files exchange between the components is not suitable for many use case scenarios. This feature is highly required for semi-realtime distributed applications e.g. in the bio-medical domain or in online video processing and analysis.

In WS-VLAM a workflow is composed not from particular Grid jobs or services but from components with special interface. These components are called modules, they are the core entities of the WS-VLAM data driven workflow. Thus a module can represent a specially developed application which uses the WS-VLAM native module API (libvlport), a web service or a legacy application.

The runtime control of the execution of a distributed workflow provides ability to monitor the execution and influences the behavior of workflow components. WS-VLAM supports several ways of runtime control: direct interaction with the user interface of a module (remote X GUI access) and module parameter control (reading flags and values set by a module and updating these values from outside the module). Monitoring delivers all the log data from remote modules to the WS-VLAM user interface thus all the issues in module execution can be tracked centrally.

Intensive distributed data processing might take a long time. To facilitate the handling of the executing workflow, the system provides capability to close the user interface and detach from the workflow engine and re-attach later during runtime.

Thus the core features of the system we present are:

- Dataflow is used as a driving force, but not a control flow;
- Generic components as workflow entities: WS-VLAM modules, which represent either specially developed software components in C++/Java/Python using WS-VLAM API, or interface legacy applications and web-services;
- Distributed execution: support for Grid job submissions together with web serivces and local tasks

within a single workflow;

- Support for legacy applications wrapped as modules (flexible XML configuration);
- Support for remote graphical output (remote X display for Grid jobs is provided);
- Interactivity support (online control via parameters, and via remote graphical output);
- Decentralized handling of intermediate data;
- Decoupling GUI and engine;

2.2 The Model

What characterizes a support environment for scientific experiments is its underlying experiment model. Among others, the experiment model defines how the functionality of the support environment is presented to the users, how scientists interact with the environment, and how experiments as well as experimental data / information are stored and managed. Furthermore, it allows a methodological definition and modeling of complex experiments in a problem domain. Such a methodologocal definition in turn allows on one hand to automate steps in experiments routinely performed, on the other hand to transfer of the experimental knowledge of domain experts to novice users.

This chapter describes the experiment model adopted by WS-VLAM. However, first an illustrative example experiment is presented, which will be used when describing the experiment model.

For a detailed description of both the approach used for modelling scientific experiments and the experiment model, please refer to [**?**] FIXME.

2.3 An Illustrative Experiment Example

The example experiment is based on a simplified version of microarray experiments in the biology domain. In this example experiment, a microarray containing a number of clones is hybridized with mRNA probes that are extracted from a sample. Every clone that is spotted on the microarray represents a gene for an organism. The result of the hybridization is the hybridized array. The hybridized array is scanned by a laser scanner to obtain an image of the hybridized array. This image consists of spots with different color intensities, where a spot corresponds to a clone on the microarray. The image is analyzed using an image analysis program to quantify the intensities of the spots. The result of the experiment is this collection of intensity values for each spot.

An illustration of microarray experiments can be found at:

2.4 Experiment Components

During a typical scientific experiment:

- *input* is taken, where the input can be a physical entity (e.g. a microarray) or data (e.g. image of the hybridized array);
- an *activity* is performed on the input, where the activity can be a laboratory activity (e.g. hybridization), an instrumentation (e.g. scanning the hybridized array), or a computational process (e.g. analyzing the image of the hybridized array);

- *output* is generated, where the output can be a physical entity (e.g. hybridized array) or data (e.g. intensity measurements for spots on the microarray).

Therefore, experiments are composed of three kinds of *experiment components*, corresponding to the different types of input, activities, and output (see Figure 2.1):

1. *physical entity* (typically a part of the input to an activity in an experiment as well as a part of the outcome of an activity in an experiment),
2. *activity* (typically laboratory activities, instrumentations, and computational processes involved in an experiment), and
3. *data element* (typically either used as a part of the input to an activity in an experiment, or generated as a part of the output of an activity in an experiment).

Figure 2.1: Different types of experiment components

Physical entities are used during laboratory activities or during instrumentation. Figure 2.2 depicts the experiment components involved in the example microarray experiment. In this example, *microarray*, *hybridized array*, *sample* and *mRNA probe* constitute the physical entities.

Activities may represent laboratory activities, instrumentations, or computational processes. Examples of activities in Figure 2.2 are, respectively, hybridizing a microarray with mRNA probes from the treated sample (i.e. *hybridization*), scanning the microarray with a laser scanner device (i.e. *array scanning*), and analyzing the array image using special analysis software (i.e. *array image analysis*).

Data elements in turn correspond to both raw data generated by instruments and used as input to computational processes (e.g. *array image* in Figure 2.2), as well as processed data and/or information generated by computational processes (e.g. *array measurement* in Figure 2.2). Generic descriptive elements constitute a special type of data elements. The generic descriptive elements represent components that are common to all experiments and provide descriptive information about them. Such descriptive information does not change from one experiment to another. In Figure 2.2, *clone*, *gene*, *organism*, *scanner* and *image analysis program* are generic descriptive elements. For example, scanner represents the device used to scan microarrays and produce images. This generic descriptive element provides descriptive information about the device, such as its vendor, model, parameters, etc. Usually this information is provided once (e.g. by a domain expert or administrator), and used by scientists in their microarray experiments.

Figure 2.2: Experiment components for the example microarray experiment

2.5 WS-VLAM Experiment Model

The WS-VLAM experiment model consists of three main components [5], namely *process flow template* (PFT), *study* and *topology* (see Figure 2.3).

An experiment is defined by its template and described by its study. A *process flow template* defines the approach taken to solve a particular scientific problem, by defining the experiment components that are typically involved in the experiments of the same type (i.e. experiments that address the same scientific

Figure 2.3: The WS-VLAM experiment model

problem). Thus, a PFT standardizes the experimental approach for experiments of the same type. On the other hand, a *study* describes the solution. A study is an instantiation of an PFT. It describes the accomplishment of a particular experiment, by providing descriptions of each experiment component involved in the experiment, in other words, by providing the context for each experiment component. Thereby, it provides the context for that particular experiment. Due to the need for processing of large data sets, computational processes constitute an important part of scientific experiments. The computational processes involved in a scientific experiment are collectively called as the *experiment's topology*.

Components of the experiment model are described in details below.

2.6 Process Flow Template (NOT YET IMPLEMENTED)

A *process flow template* (PFT) defines a particular way of accomplishing an experiment of a certain type. It represents the (steps of the) experimental approach taken by a scientist to solve a specific scientific problem. In other words, a PFT defines, step-by-step, how to make a certain type of experiment. The complete definition of an experiment includes, among others, the steps involved in the experiment, their order, and the control variables and parameters as well as the protocols to be applied during the experiment. The PFT is an abstract description of the workflow to be executed, each step in the PFT will be mapped at the execution phase to a real activity bounded to a real resources.

The same type of experiments typically involve the same components (i.e. activities, physical entities, and data elements). Furthermore, these components appear in the same order in successive experiments. A PFT is composed of elements that correspond to such experiment components. A PFT element is not an actual experiment component, rather a placeholder for the component. There is one PFT element for each experiment component. Similar to representing experiment steps by PFT elements, a PFT also represents how the components in an experiment are related to each other through corresponding relationships among PFT elements (e.g. the order among the experiment steps). Therefore, by defining the components that are typically involved in the same type of experiments, a PFT standardizes the experimental approach for that type of experiments.

Figure 2.4 illustrates a PFT for the example experiment. As mentioned above, the elements of this PFT correspond to the components involved in the example experiment. For instance, the *microarray* PFT element corresponds to the microarray physical entity. Note that the generic descriptive elements (e.g. scanner) are similarly represented by generic descriptive PFT elements in this example PFT. As also mentioned above, the PFT includes the relationships between the PFT elements, such as the order of the PFT elements (i.e. the experiment flow), and the relationships between the PFT elements and the generic descriptive elements.

Figure 2.4: Example PFT for microarray experiments

From one point of view, a PFT can be seen as a template for a certain type of experiment, since it represents both the experiment components that are common to successive experiments and the relationships among them. However, in the WS-VLAM experiment model, a PFT means more than a template. PFTs are defined by scientists that have extensive knowledge and experience in the scientific domain of the experiments (i.e. by domain experts). Therefore, a PFT captures the expertise and knowledge of the expert in that domain, and serves as a facilitator for preventing the loss of expertise and knowledge

by transferring them to the users of the PFT. This way, PFTs can be used to provide assistance to novice users for complex experiments, helping them avoid making mistakes and increase the efficiency and accuracy of their experiments.

2.7 Study (NOT YET IMPLEMENTED)

A study is an instantiation of a PFT. While a PFT defines the experimental approach taken to solve a scientific problem, a *study* represents the solution. It describes the accomplishment of a particular experiment by describing each of the components involved in that experiment. Descriptions of the experiment steps include the actual values that are used for the variables and parameters, and any other information that is necessary to describe the activities performed, physical entities that are treated/handled, or data elements that are used or generated during the experiment. In other words, a study provides the context for each experiment component, and hence the context for the entire experiment. Each component composing a study is an executable task, the mapping form the abstract description of the PFT element to the real and executable services available is performed by a the matchmaker which performs the mapping based on the semantic annotation of the VL-e evalable services, here a service can be a simple web service or an entire workflow.

The example study depicted in Figure 2.5 describes a particular microarray experiment to study the expressions of genes in a mouse tissue. It is an instantiation of the example PFT given in Figure 2.4. In this example study, the PFT elements are instantiated, i.e. descriptions of the experiment components involved in this particular microarray experiment are provided. Due to its large size, details of some study elements (namely hybridization, hybridized array, array scanning and hardware) are omitted from the figure. Similar to the example PFT, this example study also includes elements corresponding to the generic descriptive elements (e.g. clone, gene, organism). Since a microarray contains thousands of clones and there is one gene for each clone, the study elements for clones and genes are depicted in multiple copies. Also similar to the example PFT, this study provides the relationships between study elements (i.e. the experiment flow), and between study elements and generic descriptive study elements.

Figure 2.5: Example study for a particular accomplishment of the example microarray experiment

2.8 Concrete Workflow

Large size of generated data is one of the main characteristics of experiments and applications in e-science domains. Availability of a high-performance hardware and software infrastructure for the processing and analysis of these large data sets is among the major user requirements. Therefore, computational processes constitute an important part of scientific experiments, and hence require special attention. This section focuses on the computational processes and addresses their representation as part of the WS-VLAM experiment model. For simplicity, 'computational processes' will be referred to as 'processes' in the remaining of this subsection. Furthermore, existence of a software tool is assumed for each computational process (e.g. ScanAlyze for array image analysis).

In an experiment, data flows from one process to another during its processing and analysis (e.g. from a clustering process to a visualization process). This data flow can be represented by a directed graph. Nodes in the graph are processes, while the connecting arcs represent the data flow through the processes. This data flow graph represents the experiment's computational processing, hereafter referred to as the *topology*. A topology consists of a number of processes connected to each other. A process may be performing an experiment specific task or a generic task. An example of experiment specific tasks is the control of a laboratory instrument, while file manipulation is an example of generic tasks. When

defining a topology, a user specifies the values for the required variables and/or parameters for each process. Since there is a software tool corresponding to each process (hereafter referred to as *modules*), the variables and parameters are actually the variables and parameters required by the module. When defined, processes in the topology are executed on one or more computers.

Figure 2.6 depicts a topology for the example microarray experiment, which consists of three processes, each with a corresponding module. This example topology aims to analyze the image of the hybridized microarray and quantify the intensity values of the spots on the microarray. For this purpose, the first process reads the image file from its location and transfers the image to the analyzer process, which quantifies the spot intensities and passes the intensity values to the last process. The last process writes the analysis results into a new file. File reader and file writer processes are examples of generic processes, while image analyzer is an experiment specific process.

Figure 2.6: Example topology for the example microarray experiment

A (computational) process is a specific kind of activity, hence, it is considered as an experiment step. Therefore, a PFT may contain elements that correspond to processes. Consequently, particular experiments that are instantiating this PFT contain descriptions of these processes in their studies. This is illustrated in Figure **??**, where the PFT, the study and the topology for the example microarray are provided. Note that the figure contains the same PFT, study and topology for the example experiment that were given in Figures 2.4, 2.5 and 2.6, respectively. As can be seen in this figure, both the PFT and the study include elements that correspond to computational processes, which are encircled (i.e. array image analysis process and its module). Furthermore, the PFT and the study also contain elements that correspond to the input/output for the computational process, which are printed in bigger fonts (i.e. array image as input and array measurement as output). This topology consists of three processes, corresponding to input generator (i.e. file reader reading the array image), process (i.e. image analyzer processing the array image), and output consumer (i.e. file writer storing the array measurement raw data). Also note here that the study and topology of an experiment complement each other to provide information about that particular experiment. For instance, location of the array image that is required by the file reader is contained in the study (in specific, in the study element for the array image), while the topology provides run-time information (e.g. the needed environment variables and their values).

Figure 2.7: PFT, study and topology for the example microarray experiment. Details can be found in Figures 2.4, 2.5 and 2.6

2.9 The WS-VLAM Composer

To allow the manipulation of complex scientific processes, the WS-VLAM environment provides a user interface that will guide the WS-VLAM end-user and hide all the unnecessary details. The WS-VLAM user interface consists of three main components that can be invoked to perform specific tasks. The three components are the PFT editor, PFT Mapper, and the workflow composer.

- The PFT editor (NOT YET IMPLEMENTED): is used only by application domain experts. The PFT editor allows the domain expert to define the process flow template the experiment to be executed within the WS-VLAM environment. The PFT can be instantiated as many times as needed by the WS-VLAM end-users. PFT editor allow user to create and describe a workflow on abstract tasks with the needed dependencies.

- The PFT mapper (NOT YET IMPLEMENTED): is used by the WS-VLAM end-users to create an excutable instance of existing PFT (the Study). Within the PFT mapper the WS-VLAM end-user is guided through this study using context-sensitive interaction. The WS-VLAM Mapper tries to map evry step in of the PFT into a concrete executable task. To achieve this goal the PFT Mapper will use semantic matching bewteen the abstract description provided by the PFT and the semantic annotation provided services available and accessible by the end-user.
- The workflow composer: is used by the WS-VLAM end-users to define the processing elements composing his experiments. In the context of WS-VLAM environment, an experiment is represented by a data flow graph (DFG). This DFG usually contains experiment specific software entities as well as generic software entities (referred to as WS-VLAM Modules). The WS-VLAM module to be executed can be selected from a predefined list of processing elements.
- There is also the web user interface which allows the just Browsing the service repository and executing application workflow.

2.10 The WS-VLAM Run Time System Manager (RTSM)

This system concerns only the module developers and some of the core developers. WS-VLAM RTSM is used for submitting jobs to Grid enabled clusters on your behalf, using your Grid credentials.

More details on the WS-VLAM concepts can be found in WS-VLAM publications
available at: http://www.science.uva.nl/ gvlam/wsvlam.

Chapter 3

Getting and Installing WS-VLAM Software

In the following subsections, all the steps needed for getting, installing and running the WS-VLAM software are presented in detail.

3.1 Overview of WS-VLAM releases

Currently, there are two ways to use the WS-VLAM software:

- The **WS-VLAM-site** distribution which is currently deployed on the DAS3 cluster at UvA (frontend host: fs2.das3.science.uva.nl). Logon on one of the frontend hosts and use the WS-VLAM environment directly. The site distribution consists of two GAR files and the client Bundel.
- The **WS-VLAM-client** distribution which can be installed on a local windows or linux computer to access a remote GRID cluster which has a site distribution installed. Currently it is not possible to use the client on a Microsoft windows environment.

If you are not sure which distribution to use, please contact `gvlam@lists.vl-e.nl` and just ask.
Also, check the READMEs located in the distributions for the latest information.

3.2 Getting ready to use the WS-VLAM software

In order to use the WS-VLAM software, you have to meet the following requirements:

- Valid grid credentials: a certificate signed by the Dutchgrid Certification Authority. If you don't have one, you can request one by following the procedure explained at the following site:

  http://certificate.nikhef.nl/userhelp.html

  Note: the procedure of getting the signed certificate from the certification authority may take a few working days.

- An account on the DAS3 super computing cluster: send an email to:

  `das-sysadm@cs.vu.nl`

  with the subject:

  '`Request das3 account`'

  Also specify in this email that you want to be added to the gridmap file since you will use grid resources.

- For an account on the matrix cluster: send an email to:

  `bouwhuis@sara.nl`

  with the subject:

  '`Request matrix account`'

  Also specify in this email that you want to be added to the gridmap file.

Without meeting the above requirements, you really can **NOT** use WS-VLAM software.

Note: WS-VLAM core developers: please ask for a CVS account by sending an e-mail to `gvlam@lists.vl-e.nl` with the subject: '`request CVS account`'

Software requirements:

- Sun Java 2 Runtime Environment, Standard Edition (build 1.5) or higher.

For local client:

- A secure shell client with enabled X-tunneling to login from your local desktop on the remote grid cluster.

3.3 The WS-VLAM client distribution

The WS-VLAM client allows you to run the client part of the WS-VLAM toolkit on your machine. The WS-VLAM client is a java program and thus can be used on both Microsoft Windows and Linux operating systems with the proper Java Runtime Environment (see section 3.2).

3.3.1 Starting WS-VLAM using the client distribution

If you want to use the shared or site distribution (for example the one on fs2/DAS3 or the Matrix clusters at SARA), skip this section read the next section.
In order to get, install and set it up on your machine, follow these steps:

- Copy your Grid credentials to your machine in the following directory:

Linux: under `$HOME/.globus`

Windows: we suggest under `My Documents\globus`

We will refer to this location as CERT_HOME.

- Download a copy of the `$vlam-client-<version-number>.tar.gz` file from the WS-VLAM download page at: http://www.science.uva.nl/ gvlam/wsvlam or from the WS-VLAM installation on fs2:

  fs2:/home/vleroot/wsvlam/stable-<ersion-number>

  Linux: untar the file by typing in:

  ```
  $ tar -zxvf vlam-client-<version-number>.tar.gz
  ```

  This will create `$ vlam-<version-number>` directory in the directory that the above command is excuted.

  Windows: Double click on the file and untar it to a desired location. The file will be extracted into `$ vlam-client-<version-number>` directory, in that location.

  Now go in to this directory. We will refer to this location as VLAM_HOME

- Configure the globus commodity toolkit.

  Linux: not applicable.

  Windows: to do this, you need your globus credential `usercert.pem` and `userkey.pem` and the trusted CA authorities, this can be found in the `/etc/grid-security/certificates/` on the systems where globus is installed (it is better to check with your local globus administrator).
  After this you can configure the globus credentials, you can double click on the `cog-jglobus.jar` icon in the **VLAM_HOME**`/lib` directory and follow the instructions.

- Configuration of WS-VLAM client is performed automatically when you start the first time the WS-VLAM compose. A configuration window will be displayed and you will be requested to set the path to the following elements: globus loation, the WS-VLAM library, the workflow components descriptions, the URI of host where the WS-VLAM service are running

  Default value are

  - the globus location= **VLAM_HOME**`/composer/lib/auxTools/gt4.1`
  - the WS-VLAM library= **VLAM_HOME**`/composer/lib`
  - the workflow components descriptions = **VLAM_HOME**`/modules`
  - URI https:www.pc-vlab19.science.uva.nl:8443

  you can always change later the WS-VLAM client configuration by using the configuration item in the File Menubar of the WS-VLAM composer window

Note: When accessing the grid from a remote machine, you'll have to install your globus certificates in your local environment and install them into the he account of the remote cluster you are using.

More information about how to use WS-VLAM GUI can be found in section 5.

3.4 WS-VLAM site distribution

The deployment of standalone site distribution, you need to have Globus container installed on your machine. It will be enough to downlaod on the WS-CORE of the GLobus middleware

(http://www.globus.org/toolkit/downloads). Then you have to delpoy the two GAR files provided in the distribution:
`nl_wtcw_vle_rtsm.gar` and `nl_wtcw_vle_repository.gar`
to do that use the globus deployment command `globus-deploy-gar` for more information about gar deployement. This command will copy the GAR file to the appropriate directory of the globus container (http://www.globus.org/toolkit/docs/4.0/common/javawscore)

Note: If the ws-core of GT4 has been installed by your system administrator it is likely that the access to the deployment directory is restricted. Ask him to deploy the GAR files for you.

For information about setting up a site distribution, contact `gvlam@lists.vl-e.nl`

Next, we will describe how to start WS-VLAM client on a machine with a shared/central WS-VLAM installation. The installation on fs2 will be used as example.

3.4.1 Starting WS-VLAM using a (shared) site distribution

This way of starting the WS-VLAM client is recommended for end-users as well as demo users. The installation on DAS3 (fs2.das3.science.uva.nl) is used as example.
To start this frontend or client, please follow these steps:

- Start a secure remote connection using ssh to DAS3 by typing in and executing:

  ```
  $ ssh <username>@fs2.das3.science.uva.nl
  ```

  If available, tutorial users can make use of the WS-VLAM demo account, which allows you to run some of the WS-VLAM demos. In this case you should login as `vledemo` by:

  ```
  $ ssh vledemo@fs2.das3.science.uva.nl
  ```

- Ask your system admin for the WS-VLAM installation.

  On DAS3 it is located at `$ /home/vleroot/wsvlam/stable-<version-number>`
  We will refer to as VLAM_INSTALL. This variable might already be set. To check this type the following command:

  ```
  $ echo $VLAM_INSTALL
  ```

  If it is set you can use this variable to locate your WS-VLAM installation.

- To start your WS-VLAM worklflow composer, go to VLAM_INSTALL location and type the following command:

  ```
  $ ./startComposer
  ```

  or if VLAM_INSTALL is defined:

  ```
  [fs2]$ $VLAM_INSTALL/startComposer
  ```

  or use the full path (for example on fs2) to the WS-VLAM installation as follows:

```
[fs2]$/home/vleroot/wsvlam/stable-<version-number>/startComposer
```

Follow the instructions to create your local WS-VLAM profile.

This command starts the WS-VLAM configuration GUI, which will help you set the variables specific to your environment.
In this window you can check the default settings for your environment. Change these if they are not correct. Otherwise use the defaults as follows:

VLAM_HOME as described above

VLAM_INSTALL to `$ /home/vleroot/wsvlam/stable-<version-number>`

VNC_SERVER_HOST to `fs3.science.uva.nl`

VNC_VIEWER_EXEC is not needed to be set, just leave as it is.

This command will:

Create '`.vlamrc` ' in your home directory HOME which will be used each time you start vlmain and update `$VLAM_HOME/etc/configuration`.

- Finally, you can start the WS-VLAM by executing the following command from your VLAM_HOME directory:

Linux:

```
$./startVleComposer
```

This command will first check if you already have a WS-VLAM configuration (`.vlamrc`) and start the WS-VLAM using those setting. Otherwise it will first create `.vlamrc` and `$HOME/myvlam` directory under your home directory (for storing your local WS-VLAM profile) and then will start the WS-VLAM software.

More information about how to use WS-VLAM workflow composer can be found in chapter 4.

Chapter 4

The WS-VLAM Composer

4.1 Starting WS-VLAM Composer

The following sections describe the process of starting the WS-VLAM Composer.

4.1.1 Proxy Init

When starting WS-VLAM Composer by typing startVLeComposer.sh, the first you need to check are the credentials, look at the bottom corner of the of the composer if it says "Proxy expires !!", you have to generate a valid proxy before you can execute your workflow. To do this use the button "Send credential to delagation service" in the toolbar menu. You will be aske to valid the delegation of your credential then you will be asked to supplying you grid certificate password or passphrase (see section 3.2). A proxy allows several hours of access to grid resources without retyping your password.

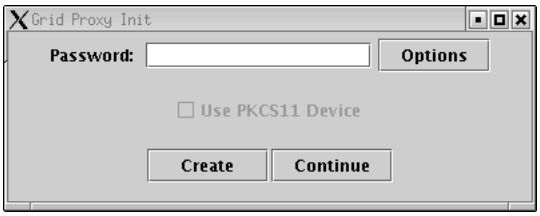

Figure 4.1: : Initialize Grid Proxy

Type in the password (See figure: 4.1) and click on the Create button, a proxy is started that keeps running for 24 hours.

Pressing Options shows the menu as seen in 4.2. In this menu you can see where you Grid Credentials are stored and how long they are valid, furthermore you can specify a longer lifetime for your grid proxy and the strength of the encryption used.

With a valid proxy, you will go directly to the main WS-VLAM Composer.

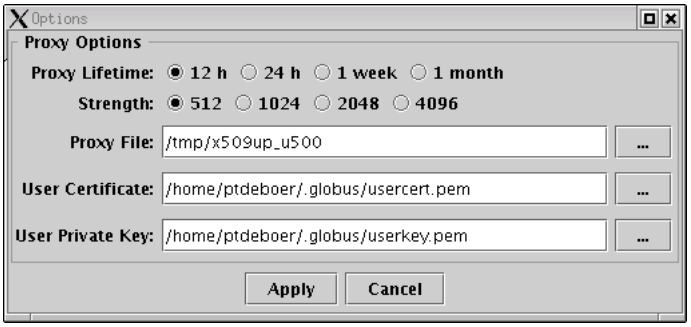

Figure 4.2: : Grid Proxy Options

4.2 WS-VLAM Composer

4.2.1 Introduction

The WS-VLAM Composer will allow you to compose a workflow using existing workflow components stored in the shared module repository, or your own local experimental workflow components.

When the WS-VLAM Composer has 4 main areas (Fig 4.3): the Workflow component palette, property panel, monitoring consol and the composition panel.

- the Workflow component palette: shows the list of available workflow components
- property panel: shows information about any selected workflow component
- monitoring consol: shows monitoring information about the execution of the workflow
- the composition panel: allow to compose a workflow by dragging components from the Workflow component palette to the composition panel

If there are local workflow components in the locale module directory (for example in your VLAM_HOME/workflow components) as explained in chapter 5 for testing in module development, these local workflow components will be loaded during the startup of composer.

If during testing there are additional workflow components created or some modifications are performed and the user needs to rescan the local workflow components, a method to do this is provided under the Tools menu as shown in Fig **??** TODO.

workflow components can be created by dragging a component from the list shown in the Workflow component palette and dropping it into the composition area. To do this, first select a component from the list and then press the left mouse button on top of it. Hold the button down and move the mouse pointer over the composition area. Finally, release the button and a figure looking similar to the ones in the graph area in Figure 19 should appear. If nothing happened, try again, making sure not to release the button before hanging the mouse pointer over the graph. This may be done for any of the workflow components that are in the list. workflow components are removed from the list by selecting them and pressing the 'delete' key on the keyboard.

To create a link between two ports, hold the mouse pointer over a port on a workflow component in the composition area and press the left mouse button. Hold the button down and move the pointer to another port. Releasing the button now should result in a line between the two ports (as shown in Figure 4.4).

Note:

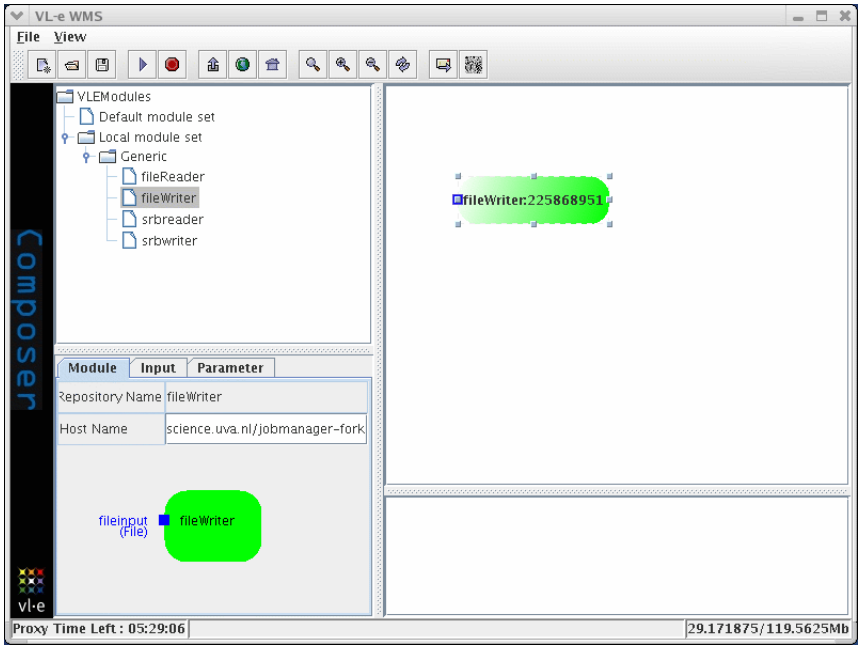

Figure 4.3: The WS-VLAM composer

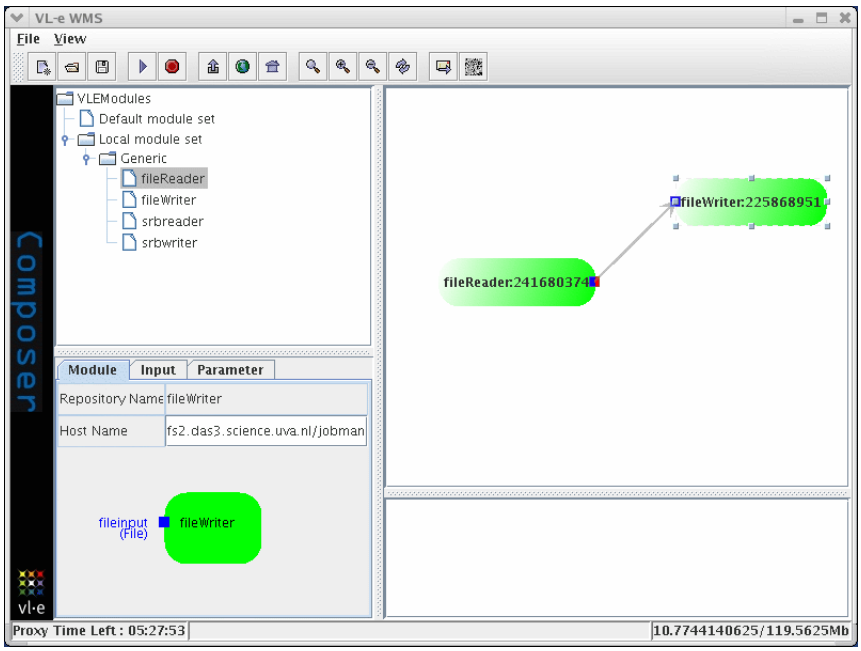

Figure 4.4: Composing the experiment

The user can only create connections from an outgoing port (blue) to an incoming port (red) and only if they are of the same type. The names and types of the ports are written in the property panel before the incoming and after the outgoing ports. The selected component in Figure 20 has one outgoing port, 'port0', with type 'File'. The port it is connected to must therefore be an incoming port also with type 'File' or any other type derived from the Filetype like 'File/SRB' or 'File/'.

In order to execute an experiment, certain values must be filled in beforehand. It is up to the user to make sure that this has been done correctly. Anything that might have to be filled in can be found in the pop-up menu of the component or in the property panel.

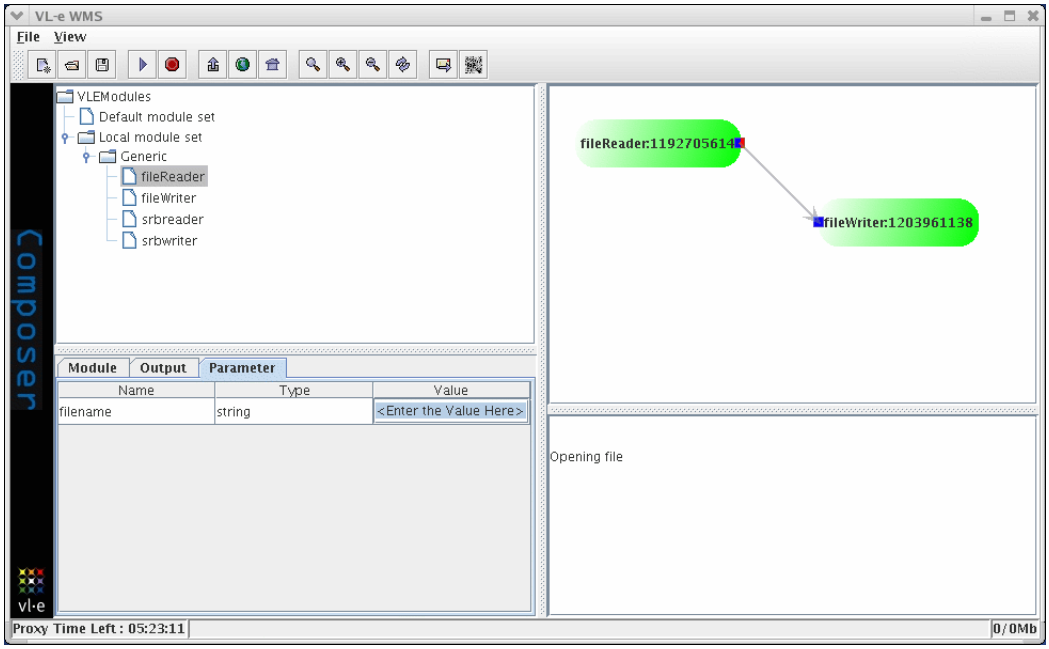

Figure 4.5: set up the run time information

An experiment may be executed by clicking on the button 'Run' in the toolbar menu. A Monitoring window will appear with that contains a tab for each component composing the workflow in which you can see 3 tabs: 'Std Out' for the standard output, 'Std Err' for the standard error, and "Module status' for the status of the component (see Figure 4.6).

At bottom of the monitoring window, there are two buttons a 'Get ...' button and 'View Graphical Output' button. The get button will allow to subsribe to receive events related to the active tab i.e. if the active tab is the 'std out' pushing this 'Get Standard Output' button will allow you to see the standard output of the component regardless where it is currently executed. The 'View Graphical Output' button allow you to see the graphical output of the selected component if it has one.

The 'Defaut' tab of the monitoring window is initialised to receive event related to the status of the entire execution of the workflow. You see there messages related to submission of the credential, the workflow and the termination of the worflow.

4.2.2 The toolbar menu

New

This will clear the WS-VLAM composer of all workflow components.

Load

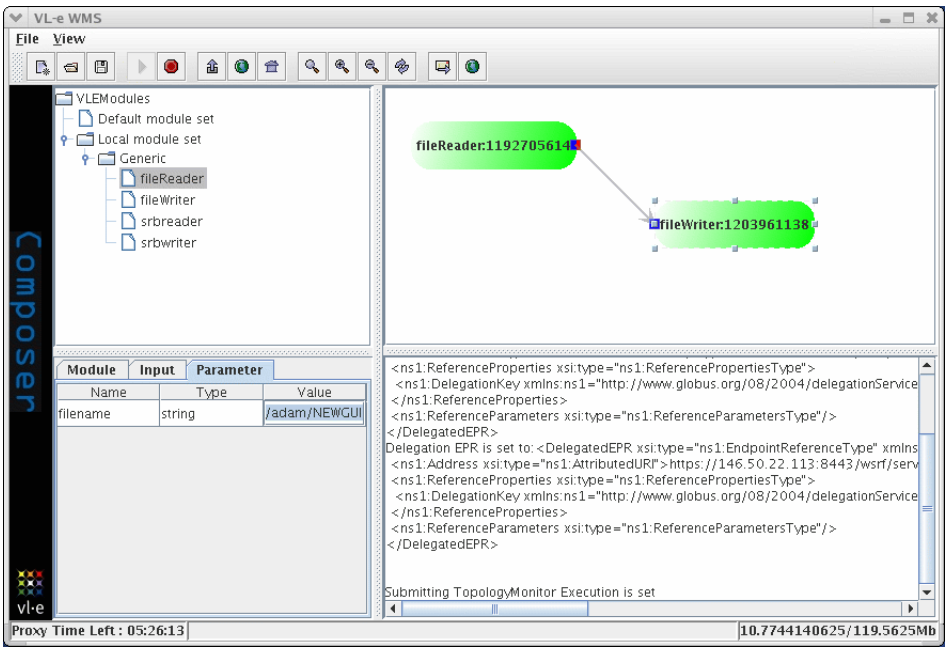

Figure 4.6: Running the experiment

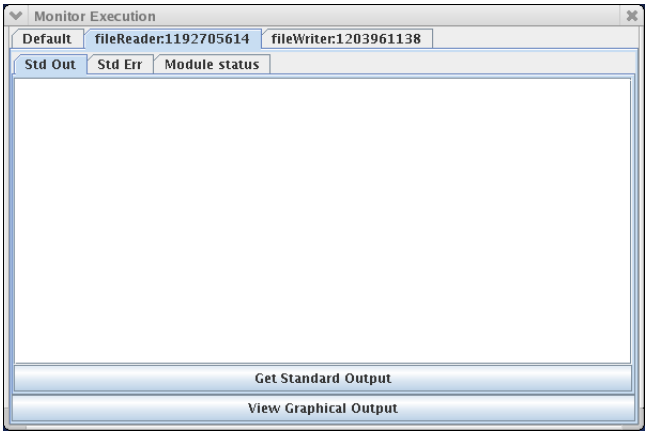

Figure 4.7: Monitoring workflow

A new window will appear with a list of available wokflow. Select a workflow from the list and click on the 'OK' button to open it. Clicking on the 'Cancel' button will remove the window and return to the composer.

Save

This will save the current workflow in the local file system.

Run

Runs the workflow described by the current composer.

Destroy

Destroys any current active exection of the worklfow shown in the composer

Create Composite Module

Groups the components composing the worklfow shown in the composer as one component called composite component.

Load Web Service

Parses a given WSDL and convert each operation composing the interface into a workflow component that can be used within the WS-composer

Load VLe Virtul Browser

Opens the Virtual Browser (drag and drop facility is possible between the Virtual Browser and the all text area in the composer.

Zoom normal

Shows the worklow in its normal size

Zoom in

Magnifies the size of the worklfow shown in the composer

Zoom out

reduce the size of the workflow shown in the composer

Correct layout

Tries to find a better layout for the workflow shown in the composer (limiter feature using only simple layout algorithm)

Send Credential to delegation service

Allow you to send you Grid credential to delaegation service on the server side to be used at the execution phase to access grid resources.

Assign delegated credential

Allow users which does not have Grid credentials to use third party credential.

4.2.3 The Edit Menu

New

This will clear the WS-VLAM composer of all workflow components.

Load

A new window will appear with a list of available wokflow. Select a workflow from the list and click on the 'OK' button to open it. Clicking on the 'Cancel' button will remove the window and return to the composer.

Save

This will save the current workflow in the local file system.

Quit

Exits the composer, you will be asked for confirmation in a pop-up window, if you have unsaved changes a warning will also be displayed in this pop-up.

4.2.4 The View Menu

Zoom normal

Shows the worklow in its normal size

Zoom in

Magnifies the size of the worklfow shown in the composer

Zoom out

reduce the size of the workflow shown in the composer

Chapter 5

The Modules

5.1 Introduction

The essential component of the WS-VLAM environment is a WS-VLAM module. WS-VLAM modules are data processing blocks for building distributed applications. All application functionality is contained within these modules. Multiple modules can be connected to each other to form a VL Experiment topology. To communicate with each other modules use input and output ports. A module can have more than one input port (or none at all) and more than one output port (or none at all). The communication via these ports is the only way for modules to exchange data within the WS-VLAM environment. Modules can be implemented in several languages. The programming languages currently supported by WS-VLAM are `C++` , `Python` and Java.

From a developer's point of view WS-VLAM modules are applications that use a special library provided by the WS-VLAM framework: the "`vlport`" library. Communication ports in this library are represented by streams. In `C++` they are standard `C++` compatible STL streams. All a module developer has to do is to create input and output ports and implement the functionality of the module. The framework will take care of the connections within the WS-VLAM environment. The following sections describe the necessary steps to create a WS-VLAM module.

5.2 Module developer guide for `C++` developers

5.2.1 WS-VLAM module structure

WS-VLAM modules are created using object-oriented methods. A module is represented by a `C++` class that inherits from `VL::VLApplication`. A very simple template for a module looks like this:

```
#include "vlapp.h"
#include <fstream>

class MyVLApplication : public VL::VLApplication
// user module inherits from VL::VLApplication
{
  // constructor, destructor and definition of ports must
  // be here, as described later

  int vlmain(int argc , char **argv)
```

```
  {
    // main module functionality is placed here
  }
};

int main(int argc , char **argv)
{
  // initialization of Globus modules
  globus_module_activate(GLOBUS_COMMON_MODULE);
  globus_module_activate(GLOBUS_IO_MODULE);
  {
    try {
      // create user module
      MyVLApplication app(argc, argv);
      // execute user module
      app.run();
    }
    // catch exceptions generated by vlport framework
    catch(VL::Exception *e)
    {
      std::cerr << "Exception: " << e->what();
      delete e;
    }
  }
  // deinitialization of Globus modules
  globus_module_deactivate(GLOBUS_IO_MODULE);
  globus_module_deactivate(GLOBUS_COMMON_MODULE);
  return 0;
}
```

Here the `main()` function does little more than executing procedures to initialize the Globus environment. Once that is done it starts the code for the WS-VLAM module. The module's main functionality should be contained in the `vlmain()` method of the derived class. This method is pure virtual in the base class and therefore has to be implemented. Note that all library classes are placed into namespaces. (Most of them are in the VL namespace, see below). Therefore you have to specify this namespace before the name of a class or import it by `using namespace VL` statement. As you can see from this code command line arguments are also passed to `vlmain()` [1].

In the template shown above the module is not able to communicate with other modules. To be able to do that the module must first define communication ports.

5.2.2 Communication with other modules

To communicate with other modules a module must define ports. The ports in a WS-VLAM module behave like standard `C++` I/O streams.

Defining communication ports The ports are presented as instances of the `VL::vistream` or `VL::vostream` classes for input or output ports respectively. Every port must have a unique name represented by a character string. The ports must be created in the constructor of the module either using `createDefaultIPort(char *portname)` to create an input port or `createDefaultOPort(char *portname)` to create an output port. Likewise the ports should be deleted in the class destructor. Thus the code should look like:

[1] Although you can't directly tell by looking at this code it may be useful to know that in the `VL::VLApplication` constructor these parameters are passed to the CORBA initialization function. It is therefore possible to specify CORBA initialization parameters from the command line. See omniORB developer guide for details.

```
class MyVLApplication : public VL::VLApplication
{
  public:
    // constructor
    MyVLApplication(int argc , char **argv) :
        VL::VLApplication(argc, argv)
    {
      // create input port called "port_in"
      myInputPort  = createDefaultIPort("port_in");
      // create output port called "port_out"
      myOutputPort = createDefaultOPort("port_out");
    };

    // destructor
    virtual ~MyVLApplication()
    {
      delete myOutputPort; // destroy output port
      delete myInputPort;  // destroy input port
    };

    int vlmain(int argc , char **argv)
    {
      // main module functionality placed here
    }

  private:
    // input and output port stream
    VL::vistream  *myInputPort;
    VL::vostream  *myOutputPort;
};
```

Using communication ports Once created the ports can be used as standard C++ input or output streams [2]. All operations applicable to standard C++ streams can be applied to the ports. For example, the operator "<<" and ">>" can be used to output and input data e.g.:

```
...
int vlmain(int argc , char **argv)
{
  int i=100;
  *myOutputPort << i << std::endl << std::flush;
  *myInputPort >> i;
};
...
```

Note that we explicitly add an end-of-line and a flush token to the output port; this is to flush the buffers of the output port explicitly thereby making sure the message is transferred. Under normal circumstances actual communication will only take place when an internal communication buffer is full. This communication buffer is to improve communication efficiency. Note also that streaming input over ports is "blocking"; if no input is available on the stream program execution will halt until there is new data available. This mechanism follows the rules for C++ standard input streams. Refer to section 5.2.5 for information on how to do non-blocking input with a VL input port. *Note that there is no full support for non-blocking input in VL yet.*

[2] Please refer to a C++ textbook for a more detailed description of C++ streams.

Here is another example of what the main module functionality could look like. Here we receive a file from an input port and store it in `/tmp/file.dat`.

```
...
int vlmain(int argc , char **argv)
{
    // open file for writing
    std::ofstream fstr("/tmp/file.dat");
    // read data from input port and store it to the file
    fstr << myInputPort->rdbuf();
};
...
```

Here we open a file and fill it with the data received from the input port. We can use `"<<"` and `">>"` for standard `C++` datatypes like `integer`, `string` and `float` with input and output ports. To write raw data we can use the `std::ostream::write(const Ch* p, streamsize n)` method, to read data we can use `std::istream::get()` or `std::ostream::read(Ch* p, streamsize n)`. All these methods including the one in the example are standard for `C++` streams. A very helpful guide can be found in [1], chapter 21.

5.2.3 Compiling your modules

The template code shown in section 5.2.1 would be a safe start for any module. Remember to include the `vlapp.h` header. We recommend you to use the `Makefile` shown below to build your modules. This `Makefile` sets all compiler and linker flags correctly. Just edit this `Makefile` and change the top two lines with the name of your module and the source files that compose it. Once you have done that simply type "`make`".

```
TARGET          = Hello
SOURCES         = Hello.cpp

include $(VLAM_INSTALL)/etc/vlport.mk

OBJECTS         = $(SOURCES:.cpp=.o)

.SUFFIXES: .cpp

${TARGET}: ${OBJECTS}
        $(CXX) $^ -o $@ $(VLPORT_LIBS)

.cpp.o:
        $(CXX) -c $(VLPORT_CFLAGS) -o $@ $<
```

To build your module you can also use pkg-config utility. Configuration file for pkg-config is usually located at $(VLAM_INSTALL)/etc directory. To use it you have to set `PKG_CONFIG_PATH` variable to this location. See pkg-config man page for details.

5.2.4 Serialization

Data byte ordering between different hardware architectures (like Intel and SPARC) might differ even if the word size is the same. This situation becomes even worse when word size is different. Still WS-VLAM

modules should be able to run on a combination of distributed computers with different architectures. Therefore there is a need for a "common" data representation. The process of encoding and decoding data into and from this common data representation is called "serialization".

The standard `C++` streams provide some serialization support but it is not sufficient for all purposes e.g. the default implementation of some stream operations uses plain text to serialize numbers and other non-text data. This may result in poor performance for large data sets. When performance is important the `vlport` library provides a special facility: a class that can serialize/deserialize basic datatypes to binary form using the well established XDR (eXternal Data Representation) format [3]. To use this facility we place an instance of this class between the stream and data being transferred.

The following snipplet of code shows how to use a serialization class for data transfers.

```
// Include serialization class definitions
#include "serializer.h"

Serializer::Serializer serializer;
std::string text = "Pi is";
double pi = 3.1415926;
myOutputPort << serializer << text << pi;
```

In this example `myOutputPort` can be any `std::ostream` compatible object, not necessarily a `VL::vlostream`[4]. To read data (and convert it from canonical to native format) use the following code:

```
// Include serialization class definitions
#include "serializer.h"

Serializer::Serializer serializer;
std::string text;
double pi;
myInputPort >> serializer >> text >> pi;
```

5.2.5 Non-blocking input

If a module attempts to read data from an input stream when no data is available the module will stop execution until data arrive. This is called "blocking input". This situation is not always desired e.g. consider a module that continuously generates random numbers in a user-defined range. In order to be able to change the minimum and maximum values of the range at run time we create an input port for each one. The code would look something like this:

```
int vlmain(int argc , char **argv)
{
  Serializer::Serializer serializer;
  while (true) {
    *port_min >> min;
    *port_max >> max;
    *port_out << serializer << drand48() * (max - min) + min;
  }
}
```

[3] For detailed information about XDR representation of data see [2], XDR Technical Note section.

[4] This is the reason why the object is defined in its own namespace. All other objects including exceptions are defined in VL namespace.

However when no input is available on `port_min` or `port_max` program execution will stop. Consequently the module will stop execution until data is available on *both* ports. To remedy this we change the code a little and "take a peek" to see whether input is available before we actually get it. Here is the same example as above with this capability added:

```
int vlmain(int argc , char **argv)
{
  Serializer::Serializer serializer;
  while (true) {
    if (port_min->rdbuf()->in_avail())
      *port_min >> min;
    if (port_max->rdbuf()->in_avail())
      *port_max >> max;
    *port_out << serializer << drand48() * (max - min) + min;
  }
}
```

According to `C++` standard `in_avail()` method returns the number of bytes available in the input buffer. If there is no data in the buffer then it returns `showmanyc()`, the number of "raw" bytes available in device.

A better alternative for changing an internal state of a module is "parameterization", which is described in the following section.

5.2.6 Module parameterization

A module may have parameters that are accessible both from inside and outside of `vlmain()`. The WS-VLAM framework supports a capability to set or get these parameters during runtime. There is one function to query and one to set the value of a module parameter from within a module. These functions are defined in `VL::Application` class. The prototypes of these functions are:

```
int getParameter(const std::string &name, std::string &value);
void setParameter(const std::string &name, const std::string &value);
```

The mechanism is similar to the use of environment variables: both parameter *identifier* and parameter *value* are represented by a string[5]. The difference is that parameters can be changed outside of the program as well.

Please note: Currently there is no way to get notification that a parameter has changed. The only way to get the latest value of a parameter is to poll it.

5.2.7 Exceptions

Make sure that all user exceptions are caught inside the `vlmain()` function; no exceptions should leave this function! The reason for this is that the module's `vlmain()` function is executed in a separate thread and uncaught exceptions will lead to termination of the module via the `abort()` call.

[5] You are responsible for converting a parameter value from string representation to an internal data structure and vice versa.

Some operations in the library can throw exceptions if they fail. These exceptions implement VL::Exception interface and should be caught within `vlmain()` as well.

Please note: All these exceptions are passed by pointer. So you should catch a pointer on VL::Exception instead of a reference.

5.2.8 Controlling access to modules: Access Control Lists (ACL)

When you create an input port you create a listener that will accept any incoming connections by default. The remote calling side should provide valid credentials only to pass Globus authentication. To introduce authorization and restrict access to the module you can modify an access conrol list (ACL) based on the DN field of X509 certificate [4]. The ACL is a protected member of `VL::Application`. It has two modes: WhiteList and BlackList (default). Authorization is based on a list and authorization mode: in the BlackList mode all connections with listed DN are rejected. The WhiteList mode works vise versa: connections are accepted if DN is listed.

`VL::Application` has a protected field acl of VL::Acl type. This ACL is created with the default policy (BlackList) in `VL::Application` constructor. The default policy can be changed using `VL::Acl::setPolicy(const AclPolicy aclPolicy)` method. `VL::Acl::AclPolicy` is an enumeration with two items: `WhiteList` and `BlackList`. The following code illustrates changing the mode to WhiteList and adding users allowed to connect to the ports.

```
MyVLApplication::MyVLApplication(int argc , char **argv) :
  VL::VLApplication(argc, argv) // constructor
{
  acl.setPolicy(Acl::WhiteList);

  // Add somebody who is allowed to have access to the port
  acl.insert("/O=dutchgrid/O=users/O=uva/OU=wins/CN=.....");
  acl.insert("...");
  // create input port called "port_in"
  myInputPort  = createDefaultIPort("port_in");
  // create output port called "port_out"
  myOutputPort = createDefaultOPort("port_out");
};
```

The functions `createDefaultIPort` and `createDefaultOPort` use the set `VL::VLApplication::acl` as a default access list controller.

5.2.9 Compiling and using the `vlport` library

At the heart of every VL module lies the `vlport` library. This library controls the execution of modules and the communication between modules. Normally the `vlport` library is already installed on your system together with the other VL software. However it may sometimes be necessary to recompile the library.

Getting the sources

It is quite likely that a version of `vlport` is already installed on your system together with other parts of the VL software; check the file `$VLAM_INSTALL/lib/libvlport.so` and use it if it exists.

If that file does not exist or if by some reason you need the latest version of the library you need to check out the "WS-VLAM" module from the WS-VLAM CVS server:

```
\$ cvs -d :pserver:<your-login>@cvs.vl-e.nl:/global/ices/vlab/cvs login
CVS password: <type your password>
```

```
\$ cvs -d :pserver:<your-login>@cvs.vl-e.nl:/global/ices/vlab/cvs co WS-VLAM/comps/vlport
```

Prerequisites

The `vlport` library uses a number of other software packages that must be installed on your system. Before you continue compiling `vlport` make sure that the following software is installed on your system:

- `g++` compiler version 3.x,
- Globus version 2.4.x, compiled from source, with threads (globus_io needed),
- omniORB version 4.x (see `http://omniorb.sourceforge.net/docs.html`).

Before building the library define the following environment variables:

- The path to the Globus installation directory in `GLOBUS_LOCATION`.
- The path to the omniORB installation directory in `OMNIORB_DIR`.

Once you have done that just type `"make"`.

When building has finished you will have the library `libvlport.so` and a set of test programs in the `test` directory. Some of these test programs are quite helpful and can be used to test modules without the WS-VLAM frontend.

NOTE: The vlport library is a component of WS-VLAM software. You are strongly recommended to use the same procedure for installation as for any other WS-VLAM component.

5.2.10 Complete Examples

Using the above instructions it is possible to create modules for WS-VLAM environment. The examples of such modules follow:

Module A.

```
#include "vlapp.h"
#include <fstream>

class MyVLApplication : public VL::VLApplication
{
  public:
    MyVLApplication(int argc , char **argv)
      : VL::VLApplication(argc, argv)
    {
```

```
      myIstream = createDefaultIPort("port_in");
    };
    virtual ~MyVLApplication()
    {
      delete myIstream;
    };
    int vlmain(int argc , char **argv)
    {
      std::cerr << "vlmain() is called" << std::endl;
      std::ofstream fstr("file.dat");
      time_t t1;
      time(&t1);
      fstr << myIstream->rdbuf();
      std::cerr << "Time in sec: " << time(NULL)-t1 << std::endl;
      std::cerr << "vlmain() is finished" << std::endl;
      return 0;
    };
  private:
    VL::vistream  *myIstream;
};

int main(int argc , char **argv)
{
  globus_module_activate(GLOBUS_COMMON_MODULE);
  globus_module_activate(GLOBUS_IO_MODULE);
  try
  {
    MyVLApplication app(argc, argv);
    app.run();
  }
  catch(VL::Exception *e)
  {
    std::cerr << e->what();
    delete e;
  }
  globus_module_deactivate(GLOBUS_IO_MODULE);
  globus_module_deactivate(GLOBUS_COMMON_MODULE);
  return 0;
}
```

Module B:

```
#include <vlapp.h>
#include <fstream>

class MyVLApplication : public VL::VLApplication
{
  public:
    MyVLApplication(int argc , char **argv)
      : VL::VLApplication(argc, argv)
    {
      myOstream = createDefaultOPort("port_out");
    };
    virtual ~MyVLApplication()
    {
      std::cerr << "before delete myOstream;" << std::endl;
      delete myOstream;
      std::cerr << "after delete myOstream;" << std::endl;
    };
```

```
    int vlmain(int argc , char **argv)
    {
      std::cerr << "vlmain() is called" << std::endl;
      std::ifstream fstr("file.orig.dat");
      time_t t1;
      time(&t1);
      *myOstream << fstr.rdbuf();
      std::cerr << "Time in sec:" << time(NULL)-t1 << std::endl;
      std::cerr << "vlmain() is finished" << std::endl;
      return 0;
    };
  private:
    VL::vostream  *myOstream;
};

int main(int argc , char **argv)
{
  globus_module_activate(GLOBUS_COMMON_MODULE);
  globus_module_activate(GLOBUS_IO_MODULE);
  try
  {
    MyVLApplication app(argc, argv);
    app.run();
  }
  catch(VL::Exception *e)
  {
    std::cerr << e->what();
    delete e;
  }
  globus_module_deactivate(GLOBUS_IO_MODULE);
  globus_module_deactivate(GLOBUS_COMMON_MODULE);
  return 0;
}
```

Module A is a data consumer, module B is a producer. The file `"file.orig.dat"` is transferred from module B to module A via the connection established by the WS-VLAM framework. In module A the received data is stored to the file `"file.dat"`. The data transfer time is also measured.

5.3 Module developer guide for python developers

The `Python` binding has been created using the SWIG toolkit that makes C/C++ libraries available from different scripting languages. The functionality of such kind of wrapper is limited due to universal nature of the tool. But it is sufficient to make simple prototypes using `Python` as a "glue" language.

5.3.1 Building `Python` wrapper

Before you build the wrapper make sure that the SWIG toolkit and `Python` are installed in your system. The SWIG version should be higher then 1.3.19. The wrapper has been tested on Fedora Core 2 Linux distribution with swig-1.3.19-6.1 and python-2.3.3-6 installed and Redhat Enterprise AS edition with swig-1.3.20 and python 2.2.3.

To compile the python wrapper the environment variable VLAM_INSTALL must be set. Just type ``` `make dist` ```. You will find the library in dist/lib subdirectory. The wrapper consists of an "adapter" shared library with the name "_vlport.so" and a python module named "vlport.py".

NOTE: The `Python` wrapper is a component of WS-VLAM software. You are strongly encouraged to use the same procedure for installation as for any other WS-VLAM component. After installation the wrapper can be found at `$VLAM_INSTALL/lib` directory.

5.3.2 Using `Python` wrapper

Initialization To use the library you need to load it first. It can be done by using "import" command:

```
$python
Python 2.3.3 (#1, May  7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import vlport
>>>
```

After this you can create an instance of vlport application:

```
>>> vlapp=vlport.VLAppFactory.activate(["inputPort1","inputPort2"], ["outputPort1", "outputPort2"])
```

Here we create vlapp with two input and two output ports. If the application has no input or output ports then we have to pass an empty list to `activate` function.

Using this reference we can get references to our input and output ports:

```
>>>inputPort1=vlapp.getInputPort("inputPort1")
>>>outputPort1=vlapp.getOutputPort("outputPort1")
```

We should provide the same names for the ports as specified at the initialization stage.

port access API After that we get the reference to a port which we can use to read and write data.

Here is the description of the methods associated with the output port:

writeInt(integer number) : writes an integer to the output port

writeDouble(decimal number) : writes a decimal number to the output port

writeString(String) : writes a string to the output port which accepts strings

write(String) : write a raw data from String to the port without serialization

Here is the description of the methods associated with the input port:

readInt() : reads an integer number from the input port

readDouble() : reads a decimal number from the input port

readString() : reads a string from input the port

read([size]) : Read at most size bytes from the port (less if the read encounters EOF before obtaining size bytes). If the size argument is omitted, it reads all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately.

5.3.3 Module parameterization

A module may have associated parameters accessible both from inside and outside of the program. The WS-VLAM framework supports a capability to set or get these parameters during runtime. There is one function to query and one to set the value of a module parameter from within a module. These functions are defined in the same name space where the functions for dereferencing input/output ports are defined:

```
>>>parameterValue = vlapp.getParameter("parameterName")
>>>vlapp.setParameter("parameterName", "parameterValue")
```

This couple of functions are used to get and set parameters. The first method accepts the string - the parameter name, and returns its value. The second method is used to set a parameter and accepts two strings: parameter name and parameter value.

The mechanism is very similar to the use of environment variables: both the parameter *i*dentifier and the parameter *v*alue are represented by a string. The difference is that parameters can be changed outside of the program as well.

5.3.4 Complete Example

The following code is a simple example of a file reader. It defines one output port named "outputPort", then it retrieves the file name from parameter and opens a file with this name. After that it reads the file to a string and sends it to the output port.

```
#!/usr/bin/python

import vlport
vlapp=vlport.VLAppFactory.activate([], ["outputPort"])
filename=vlapp.getParameter("fileName")
file=open(filename)
outputPort=vlapp.getOutputPort("outputPort")
fileDaata = file.read()
outputPort.write(fileData)
=vlport.VLAppFactory.deactivate(vlapp)
```

5.4 Module developer guide for java developers

A java program can be converted to a WS-VLAM module using the java wrapper. The wrapper uses the java native interface in order to give access to C++ vlport library. The wrapper creates an execution environment for your java code which makes it VL compatible. It starts JVM, executes application code, and creates input and output ports according to information in config.xml. An application can use these ports through the classes in nl.wtcw.vlamg.jwrapper package. They provide access to the libvlport library. This process is completely transparent for module developers. In the following sections we will describe the steps to convert a java program into a module for the WS-VLAM environment.

5.4.1 Building java wrapper

The java wrapper is a component of WS-VLAM software. You are strongly encouraged to use the same procedure for installation as for any other component of WS-VLAM. It is a standard component and should be built during WS-VLAM installation procedure.
The jwrapper is located in the `bin` directory of the VLAM_INSTALL. Needed libraries are located in the `lib` directory of the VLAM_INSTALL.

5.4.2 Defining communication ports

Communication ports for a java module are declared in the config.xml. It uses the XML syntax.

WS-VLAM java ports are references to the instances of the `VL::VLIPort` or `VL::VLOPort` classes for input or output ports respectively. Each port must have a unique name represented by a character string (in the XML description this name is specified inside XML tags texttt<port>. There are two types of ports: input and output. The type of the each port is specified by using either an XML tag <`output`> or <`input`>.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<config>
        <ports>

                <output>
<!-- Here we define output ports -->
                        <port>portOut</port>
                </output>
        </ports>
<input>
 <!-- Here we define input ports -->
<port>portIn</port>
</input>
</config>
```

A reference to the port needs to be created from the java code, either using `VLIPort.getInstance(String NameOfThePort)` to create a reference to an input port or `VLOPort.getInstance(String NameOfThePort)` to create a reference to an output port.

If you'll try to get a reference to an undefined port (not specified in the configuration file) then VLAPPException will be thrown.

```
// YOU MUST IMPORT THE JWRAPPER PACKAGE
import nl.wtcw.vlamg.jwrapper.*;
 ....

  try{
        // CREATE A REFRENCE TO INPUT PORT WITH NAME "portIn"
        VLIPort portIRef = VLIPort.getInstance("portIn");

        // CREATE A REFRENCE TO OUTPUT PORT WITH NAME "portOut"
        VLOPort portORef =VLOPort.getInstance("portOut");
  } catch (VLAPPException e){
        // HANDLE EXCEPTION
        ....
  }
  ...
```

5.4.3 Using communication ports

Once referenced from the java code, the ports can be used. Input ports provide methods to read and deserialize all basic java datatypes. There are methods to work with raw data also. Such kind of data (byte array) is transferred "as is" without serialization. See javadoc API description for details.

```
// YOU MUST IMPORT THE JWRAPPER PACKAGE
import nl.wtcw.vlamg.jwrapper.*;

...

    VLOPort outputPortRef =  null;
    VLIPort inputPortRef  =  null;

    ...

    // IT IS ASSUMED THAT REFERENCES TO THE PORTS ARE ALREADY TAKEN

    try{
            double rndn = 12345.12345;
            // WRITING TO THE WS-VLAM Output Port
            outputPortRef.writeDouble(rndn);

            ...

            // READ FROM THE WS-VLAM INPUT PORT
            rndn = inputPortRef.readDouble();

            ...
    } catch(VLAPPException e){
            // HANDLE EXCEPTION
            ...
    }
...
```

You need first to create a reference to the vlport. In this example we just declared two ports "inputPortRef" and "outputPortRef".

5.4.4 Create the config.xml

In order to use a java program as a WS-VLAM module with help of the `jwrapper` packages, you have to create a config.xml file. This file is used at the run-time by the wrapper. All the necessary information about your java module is there. The config.xml file defines all the parameters and options required for running your module.

The config.xml starts with the XML tag <`config`>. The file has two main sections: the first section defines the ports (located inside <`ports`> tag as shown in the listing **??**; the second section defines the options and attributes needed for the java code (located inside <`java`> tag.

5.4.5 define the java section in the config.xml

In java section following parameters are specified:

- the name of the java class to be executed;

- the classpath (optional: the value of $CLASSPATH variable is used in addition);
- any other options your java class may require.

In the following XML listing we introduce the XML tags that allow to specify all these attributes and options

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<config>
<ports>
<!-- define here the list of input and output ports -->
</ports>
  <java PropertyName="PropertyValue"> <!-- Here we can define System
 properties for jvm -->
<!-- Here an entry point to your program is defined -->
<main>
    your-java-main-class
</main>
<options>
<!-- Here we define additional options to jvm -->
<option>
    -server
</option>
</options>
<!-- optional classpath -->
<classpath>
/path/to/something.jar
</classpath>
</java>
</config>
```

WARNING: the name of the main class MUST be specified in JNI notation. It means you must use "/" instead of "." Example: nl/amolf/vle/ftms/rmi/AnalysisVleModule

5.4.6 jwraper API

The jwraper API includes seven methods for writing and reading standard basic datatypes, two methods to flush internal buffers and two methods to get/set module's parameters.

void writeBytes(byte[data)]: writes a byte array to an output port;

void writeShort(short number) : writes a short to an output port;

void writeLong(long number) : writes a long integer to an output port;

void writeInt(int number) : writes an integer to an output port;

void writeFloat(float number) : writes a float number to an output port;

void writeDouble(double number) : writes a double precision number to an output port;

void writeString(String str) : writes a string to an output port;

void flush() : prepares an output buffer to be sent over network (flush internal library buffer)

void hardSync() : prepares an output buffer and waits until it is sent to network.

void readBytes(byte[data)]: reads a byte array (raw data) from an input port. Array must be prepared before;

void readShort() : reads a short number from an input port;

void readLong() : reads a long integer number from an input port;

void readInt() : reads an integer number from an input port;

void readFloat() : reads a float number from an input port;

void readDouble() : reads a double precision number from an input port;

void readString() : reads a string from an input port.

5.4.7 Module parameterization

A module may have associated parameters that are accessible both from inside and outside a module. The WS-VLAM framework supports a capability to set or get these parameters during runtime. There are two functions available for that: one to query it's value and one to set the value of a parameter from within a module. These functions are defined in `VLApp` class. The prototypes of these functions are:

```
String VLApp.getParameter( String parameter_identifier );

void VLApp.setParameter( String parameter_identifier, String parameter_value );
```

The mechanism is similar to environment variables: both the parameter *identifier* and the parameter *value* are represented by a string[6]. The difference is that parameters can be changed outside of the program as well.

HINT: There is no way to get notification that a parameter has changed. The only way to get the latest value of a parameter is to poll it.

5.5 Legacy application wrapper

In order to allow the WS-VLAM users to integrate and use application programs which they don't have access to the source code with the WS-VLAM software, we have developed a legacy application (LA) wrapper.

Almost any application program working with files can be wrapped as a module in VL-e. Using the WS-VLAM LA wrapper, an application program can be converted into a WS-VLAM module without having to modify its source code. Application program that can be wrapped as WS-VLAM module must work with files (gets input data form files and write the output data to files). An example of such application is tar program: it gets one file(s) and produce other(s). Such an application program can be deployed in the same way, WS-VLAM modules are deployed (mode details on module deployment are given in Chapter 4 of the WS-VLAM Guide). Form the user point of view a LA program appears to the user as a normal WS-VLAM module with input and output ports which can be connected to other program to allow the creation of the dataflow between a number of modules.

[6] You are responsible for converting a parameter value from string representation to an internal data structure and vice versa.

5.5.1 The WS-VLAM LA wrapper deployment

The only step which specific to the deployment of LA program in the WS-VLAM environment, is the creation of configuration file (config.xml) which have to be created in the module base directory ($VLAM_HOME/modules). The configuration file contains the information needed by the WS-VLAM LA wrapper:

- the number and the type of input/output ports
- the command that have to be executed

The module deployment procedure is described in detail in Chapter 4 of the WS-VLAM Guide.

5.5.2 The WS-VLAM LA wrapper environment variables

The WS-VLAM LA Wrapper reads configuration file, creates input and output ports, inbound and outbound directory for input and output files. When WS-VLAM experiment is running the wrapper reads files from input ports (one file from each port), put them to inbound directory and executes LA program. It sets environment variables to instruct LA program where it can get input files, the names of the files to be processed and path to outbound directory. The name of the environment variables are

- $VL_INBOUNDDIR: defines the name of the input directory
- $VL_INPUT_FILES: defines the names of the input files
- $VL_OUTBOUNDDIR: defines the name of the output directory

The WS-VLAM LA wrapper also set $VL_OUTPUT_FILES variable to show the file names it expects from LA as output. The names of the files correspond with names of the ports in the module.

5.5.3 The Configuration file (config.xml)

The config.xml file consist of two main section: the section where the ports are defined and the command to be executed. An example of such configuration file for gzip application is provided with the WS-VLAM LA wrapper in doc directory. In this example, the data is read from the input port "input_port", the LA program gzip zips the input data. If this program is used in WS-VLAM environment the zipped data moved via the output port with the name "output_port" to the Grid node where it is going to be further processed.

sectionData Typing in WS-VLAM

5.5.4 Hierarchical data types in WS-VLAM

There are 3 basic data types supporting by libvlport library:

1. Raw data stream:

   A module developer can choose to work with row data stream. He gets a reference to an input port and reads the data like from row socket (byte by byte). This basic data type is intensively described in (chapter5, section 5)

2. Primitive data types (integers, doubles, strings, Booleans)

   Primitive data types are used if a module developer wants to read primitives from an input port (input stream) without dealing to the headers introduced by Complex Basic Datatype container. If content of the container is relatively small comparing to the header, it can introduce a big overhead if used for storing basic data types (this is the reason for having specific data types for primitives). Primitives are encoded using some standard technique i.e. XDR or CORBA (or java).

3. Complex Basic Datatype (CBD):

   This is a basic data type for any other complex data type. It is an annotated limited sequence of bytes. In practice, CBD is a container for storing metadata (header) and the actual data (body). I.e. for a generic file metadata is the name of a file and its attributes (optionally). For an image it can be height and width of an image and the graphical format. Types inherited from CBT are organized in MIME-like hierarchical tree. The header can be described in XML format (schema have to be developed). A module developer reads header from input port and interprets it depending on his application. Then he gets the reference to the actual data for further processing. The Libvlport library does not interpret header anyhow.

5.5.5 Complex Basic Datatype (CBD)

In spite of CBD is a very generic container, it has limitations. It can hold any information, but a module must know how to deal with this data. It prevents using generic base datatypes. Consider the following datatype hierarchy:

If a module claims that it supports Raster Image, it must support all JPEG, GIF and PNG. In reality such a module supports a limited set of subtypes, i.e JPEG and GIF. This situation can be even worse if we had this hierarchy when we developed our module, but later we have added a new datatype successor of Raster Image (i.e. BMP). The module claims that it supports Raster Image (implies JPEG, GIF and PNG). So, if BMP datatype is connected to such a module (it is allowed), it leads to a faulty situation.

This means that we still have to enumerate all supported datatypes (Raster Image/JPEG, Raster Image/GIF,) . It does not make a big difference with current syntactical matching we use to have in GVLAM.

A Better solution in order to provide a real datatype hierarchy, we have to move objects, not just a data. The object Raster Image shares the same interface between all implementations (JPEG, GIF, PNG). If a module claims that it supports Raster Image it must support all successors. The easiest solution can be provided in Java. The header of CBD can contain an URL to the implementation of the object. Then Java run-lime loader is used to get the implementation class from http-server and load implementation to the JVM. Java serialization can be used to represent the actual data for an object. Similar (but more complicated) solution can be used for C++ (load class implementation for a specified platform with dlopen() function and then deserialize actual data).

The CBD header can look like this:

```
<cbd>

   <type name= RasterImage/Gif   class=  my.raster.image.gif.class />
   <implementation url= http://fs2.das3.science.uva.nl/~gvlam/libgif.jar />
   <implementation url= http://fs2.das3.science.uva.nl/~gvlam/otherlib.jar />

</cbd>
```

5.6 Testing your modules outside WS-VLAM

Before you move your module to a module repository it would be wise to test it first. To do that the `vlport` software provides an application that can be used to test modules from outside the VL environment.

1. Set the environment variable `TEST_VLAPP_CLIENT` to the VL application controller program:

   ```
   $ export TEST_VLAPP_CLIENT=$VLAM_INSTALL/test/vlap_test_client
   ```

   We will use this environment variable as a shorthand for running the experiment.

2. Run the module, for example;

   ```
   $ ./Hello > ior.Hello &
   ```

   This command starts up the module in the background and redirects its output into a file; the contents of this file is used by the VL application client tester to locate the module.

3. Signal the module to start running;

   ```
   $ $TEST_VLAPP_CLIENT -i ‘cat ior.Hello‘ -m runExp
   ```

   This command tells the VL application client tester to signal our Hello module to start running.

In case you want to test a combination of two or more modules that need to communicate with each other over communication ports, you need to somehow make the connection known between these modules. This is done with the `connectRef` option, as shown in the following example session:

```
$ ./moduleA > ~/ior.moduleA &
$ ./moduleB > ~/ior.moduleB &
$ $TEST_VLAPP_CLIENT -i ‘cat ~/ior.moduleB‘ -m connectRef port_out port_in ‘cat ~/ior.moduleA‘
$ $TEST_VLAPP_CLIENT -i ‘cat ~/ior.moduleA‘ -m runExp
$ $TEST_VLAPP_CLIENT -i ‘cat ~/ior.moduleB‘ -m runExp
```

Here we connect the output port with name ”port_out” of module B to the input port ”port_in” of module A and start them running.

5.7 Testing your modules using the WS-VLAM

WS-VLAM support a local mode execution, in which you can test your modules using the GLVAM environment while these modules are still located under your home directory. It is important to note here, that testing your module in local mode can be done only, when you are working on host which has Globus install and up-and-running.

1. Package your module as described in Section **??**
2. move the created package to your local $VLAM_HOME/modules
3. The next time you start the WS-VLAM environment and come to point where you want to start the WS-VLAM topology editor, you will be notified that the modules you have just created under $VLAM_HOME have been detected. You will asked you if you want to load them to make them accessible within the WS-VLAM environment (select the ones you want to test and press the button load);

4. The module you have just created and load will be listed in the list of WS-VLAM module; they will be lister under root directory LOCAL_MODULES

When the modules have loaded successfully to the WS-VLAM working environment, you can use them in the same way you use standard module - instantiate (drag-and-drop in the topology editor), connect to other modules (either local or default WS-VLAM modules) and execute any topology composed of these local modules.

It is important to note here, that any topology created using local modules cannot be saved in the WS-VLAM information system. The local modules are only visible to you and cannot be see or used by another WS-VLAM user even within your domain. However, the WS-VLAM environment offers the possibility to save these topology locally, just to help you do more tests without having to create the same topology again and again. Following are some restrictions related to using local module and local topologies

1. if a local module is remove automatically the topologies containing instances of that module become invalid and cannot be used any more
2. Adding/removing ports, parameters, and arguments to a local module will create problems when trying to load local topology pointing to this modules,any changes to the core of the module probably have no impact on the loading process of the associated topology.

5.8 Logging

A very effective way to debug any code is to sprinkle it with commands that write informative messages to screen. In VL, however, modules could be running on remote systems. RTS will redirect all standard output and error streams to a file. The log files are accessible in `$VLAM_HOME/var` on remote side. The name of the file consist of a module name name plus some unique number.

5.9 Modules deployment

A VL module needs to be deployed before it can be executed. In the current version of the VL toolkit, the deployment of a VL module is the responsibility of the VL administrator who, after checking the submitted module, loads its description in the VL RTS database and moves the executable into the VL module repository.

All VL modules have to be deployed before the VL users can use them. The deployment procedure consists of two steps

1. Add the information related to module to the VL database. This is usually done by VL administrator using the script `VL-load` provided with the VL-toolkit (consult the VL module-loader Guide available at `http://www.vl-e.nl/documentation/`).
2. Make the module available for the VL Run-Time system. Copy the module package to the VL repository. In the current version we use the `$VLAM_INSTALL/modules/` as a repository for the VL modules.

VL modules can also be made locally available from within the VL toolkit *before* they are loaded in the VL databases by the VL administrator. This feature is meant to support VL module developers in the test phase of their VL modules.

To deploy VL modules for testing you need to copy the "module package" to the `$VLAM_HOME/modules/` directory. This will make the new VL modules available locally (for yourself only, testing mode). Module accessible by other users have to be copied to `$VLAM_INSTALL/modules/` (if you are using a pre-deployed version or site distribution of VL you probably don't have the right to copy file to the `$VLAM_INSTALL/modules/`, contact the VL administrator to deploy your modules system wide).

When you submit a module the VL Run-Time system will search for the module in your `$VLAM_HOME` installation first, then in the system-wide `$VLAM_INSTALL`.

Note, you can test the execution of the new VL modules only if the package is located on a grid-enabled host.

Module package To deploy a module it must be packaged first. Each module package should meet specific requirements. The package is a set of files and directories which has a root directory with the same name as the module name. To avoid conflicts with other modules the name should be unique (inside the repository). Java notation for packaging could be used for this purpose.

The module package root directory should contain the following elements:

- The module description file <`module-name`>`.mdf`, a file with the same name as the module name, and with suffix .mdf.
- The icon file for the module in gif form. This icon will be used when the module is instantiated within the VL environment. The icon file should be saved in a sub-directory called ‘‘`icon`’’
- The module executable. This file should be saved in a sub-directory called ‘‘`bin`’’. It must be called from the main.sh script - the module entry point.
- Dependencies file. If the module depends on other modules (e.g. shared libraries), you have to specify them in the dependencies file (list the names of modules in one line separated with spaces). The dependencies file should be named ‘‘`dependencies`’’.
- Profile. If the module needs some specific environment (e.g. path to additional libraries - `$LD_LIBRARY_PATH` or other) you have to specify these environment variables in the profile file. The profile file should be named ‘‘`profile.sh`’’. The variable `$BASE_DIR` is automatically expanded to the full path of the module. This file is very important for quasi-modules, consisting of just a libraries, not binaries. Those libraries can be added to a real module depending on them.
- The `main.sh` script. It is the entry point for your module, this script is called when the module is requested to run. In the `main.sh` script you have to call the real executable and use the ‘‘`$*`’’ as a command line parameter to pass the options from the VL Run-Time system to the `libvlport.so` (so every `main.sh` should contain at least the following command line: `$BASE_DIR/bin/`<`name-executable`> `$*` , see the module examples in the modules directory provided in the current VL distribution)

  Note that a module linked to `libvlport.so` depends on parameters it gets from the VL Run-Time system (see module the examples in the modules directory provided in the current VL distribution).
- any other files, libraries, executables which the module needs.
- If the module has manuals, they have to be put in the directory called ‘‘`manual`’’ (this directory is requested only in the final module deployment phase not in the test phase described in this section)

Note that the VL Run-Time system executes the bash interpreter during module initialization. So all scripts in the module packages should use bash notation.

Third party libraries can also be shared between modules. The libraries can be represented as module packages which have only the `lib/` directory with library files and the `profile.sh` file which defines the environment variables needed for the library.

Bibliography

[1] *Bjarne Stroustrup*. AT&T Labs, Murray Hill, New Jersey. The `C++` Programming Language, third edition. 1997.

[2] *ONC+ Developer's Guide*. Solaris 8 Software Developer Collection. `http://docs.sun.com/db/doc/805-7224`

[3] *OmniORB Documentation*. `http://omniorb.sourceforge.net/docs.html`

[4] *Request for Comments: 2459* The Internet Engineering Task Force `http://ietf.org/rfc/rfc2459.txt`

[5] *E. C. Kaletas, H. Afsarmanesh, and L. O. Hertzberger*. Modelling Multi-Disciplinary Scientific Experiments and Information. In *Proceedings of the Eighteenth International Symposium on Computer and Information Sciences (ISCIS03)*, 2003.

[6] *E. C. Kaletas*. Scientific Information Management in Collaborative Experimentation Environments. PhD Thesis. University of Amsterdam, Faculty of Science. ISBN 90-5776-121-1. 2004.