



Design and Development of Prototype Components for the Harness High-Performance Computing Workbench

A Dissertation
Submitted In Partial Fulfilment Of
The Requirements For The Degree Of

MASTER OF SCIENCE

In

NETWORK CENTERED COMPUTING,
HIGH PERFORMANCE COMPUTING AND COMMUNICATION

in the

FACULTY OF SCIENCE

THE UNIVERSITY OF READING

by

Ronald Baumann

6. March 2006

Supervisors:

Prof. V. Alexandrov, University of Reading
G. A. Geist, Oak Ridge National Laboratory

Acknowledgement

I want to thank all those who have given time, assistance and patience during the preparation of this thesis.

I would especially like to thank my advisor, G.A. Geist, for his intellectual support and continuous encouragement. I appreciate his invitation to do my thesis research at the Oak Ridge National Laboratory and the financial support that made it possible for me to accept it.

Christian Engelmann provided continuous bug-fixing on Harness software, review of my drafts, and ongoing completion of new releases. He also gave me research support and help with problem solving.

Thanks to Kai Uhlemann for his personal support and his work in setting up a test environment for my software.

Abstract

This master thesis examines plug-in technology, especially the new field of parallel plug-ins. Plug-ins are popular because they extend the capabilities of software packages such as browsers and Photoshop, and allow an individual user to add new functionality.

Parallel plug-ins also provide the above capabilities to a distributed set of resources, i.e., a plug-in now becomes a set of coordinating plug-ins. Second, the set of plug-ins may be heterogeneous either in function or because the underlying resources are heterogeneous.

This new dimension of complexity provides a rich research space which is explored in this thesis. Experiences are collected and presented as parallel plug-in paradigms and concepts. The Harness framework was used in this project, in particular the plug-in manager and available communication capabilities. Plug-ins provide methods for users to extend Harness according to their requirements.

The result of this thesis is a parallel plug-in paradigm and template for Harness. Users of the Harness environment will be able to design and implement their applications in the form of parallel plug-ins easier and faster by using the paradigm resulting from this project.

Prototypes were implemented which handle different aspects of parallel plug-ins. Parallel plug-in configurations were tested on an appropriate number of Harness kernels, including available communication and error-handling capabilities. Furthermore, research was done in the area of fault tolerance while parallel plug-ins are (un)loaded, as well as while a task is performed.

Contents

| | |
|---|------------|
| List of Figures | vii |
| List of Tables | ix |
| Abbreviations | x |
| 1. Introduction | 1 |
| 1.1. Overview | 1 |
| 1.2. Previous Work | 2 |
| 1.2.1. Parallel Virtual Machine | 2 |
| 1.2.2. Message Passing Interface | 4 |
| 1.2.3. Harness | 4 |
| 1.3. Project Description and Objectives | 5 |
| 1.4. Requirements and Key Problems | 6 |
| 2. Preliminary System Design | 8 |
| 2.1. Basic Principles and Concepts | 8 |
| 2.1.1. Communication | 8 |
| 2.1.2. Harness | 15 |
| 2.1.3. Remote Method Invocation Extension | 16 |
| 2.1.4. Fault Tolerant Design | 19 |
| 2.1.5. Today's Plug-in Technology | 20 |
| 2.2. Definition of Parallel Plug-ins | 22 |
| 2.3. Motivation and Features of Parallel Plug-ins | 23 |
| 2.4. Types of Parallel Plug-ins | 24 |
| 2.4.1. Distributed Parallel Plug-in | 26 |

| | | |
|-----------|--|-----------|
| 2.4.2. | Replicated Parallel Plug-in | 27 |
| 2.4.3. | Service Plug-in | 28 |
| 2.5. | Scientific Applications and Parallel Plug-ins | 29 |
| 2.5.1. | Monte Carlo Integration | 30 |
| 2.5.2. | Image Processing Pipeline | 34 |
| 2.6. | System Design of a Prototype Parallel Plug-in Suite | 37 |
| 2.6.1. | General Communication Aspects Regarding Parallel Plug-ins . | 39 |
| 2.6.2. | General Fault Tolerance Mechanisms Regarding Parallel Plug-ins | 40 |
| 2.6.3. | Parallel Plug-in for Monte Carlo Integration | 41 |
| 2.6.4. | Parallel Plug-in for an Image Processing Pipeline | 44 |
| 2.6.5. | Parallel Plug-in Manager | 49 |
| 3. | Implementation Strategy | 53 |
| 3.1. | Implementation and Integration Strategies | 53 |
| 3.1.1. | Programming Issues | 54 |
| 3.1.2. | System Environment | 54 |
| 3.1.3. | Parallel Plug-in Manager | 55 |
| 3.1.4. | Replicated Monte Carlo Integration Plug-in | 57 |
| 3.1.5. | Distributed Image Processing Pipeline Plug-in | 60 |
| 3.2. | Testing Strategies | 64 |
| 3.2.1. | Component and Integration Tests | 65 |
| 3.2.2. | System Test | 69 |
| 4. | Detailed Software Design | 72 |
| 4.1. | Application Architecture | 72 |
| 4.2. | Interface Definitions | 75 |
| 4.3. | Design of Components | 78 |
| 4.3.1. | Parallel Plug-in Component for Integral Computations | 78 |
| 4.3.2. | Parallel Plug-in Component for Image Processing Pipelines . . | 81 |
| 4.3.3. | Service Plug-in for Parallel Plug-in Management | 85 |
| 5. | Conclusion | 92 |
| 5.1. | Results | 92 |
| 5.2. | Future Work | 95 |

| | |
|---|------------|
| References | 97 |
| A. Appendix | 101 |
| A.1. Program Manual | 101 |
| A.1.1. Installation | 101 |
| A.1.2. Example Configuration Files | 104 |
| A.1.3. Program Execution | 106 |
| A.2. Program Output Listings | 108 |
| A.2.1. Output Listings of the Parallel Plug-in for Integration | 108 |
| A.2.2. Output Listings of the Parallel Plug-in for Image Processing | 113 |
| A.3. Source Listings | 120 |
| A.3.1. RMIX Interface Descriptions | 120 |
| A.3.2. Parallel Plug-in Manager | 136 |
| A.3.3. Parallel Plug-in for Integral Computation | 215 |
| A.3.4. Parallel Plug-in for a Image Processing Pipeline | 235 |

List of Figures

| | |
|---|----|
| 2.1. Shared Memory System | 9 |
| 2.2. Overlapping Memory Space | 10 |
| 2.3. Distributed Memory System | 11 |
| 2.4. Client/Server Model | 13 |
| 2.5. Network Topologies [Wik06c] | 14 |
| 2.6. Lightweight Kernel Design [EG05a] | 16 |
| 2.7. RMIX Framework Architecture [EG05b] | 17 |
| 2.8. RMIX Plug-in for Harness [EG05b] | 18 |
| 2.9. UML Class Diagram for a Plug-in Pattern [MMS02] | 21 |
| 2.10. Types of Parallel Plug-ins | 25 |
| 2.11. Loading of Parallel Plug-ins | 29 |
| 2.12. Integral Divided into Chunks | 32 |
| 2.13. Source Image | 34 |
| 2.14. Processed Image | 34 |
| 2.15. Image Processing with Copied, Fully-equipped Processing Units | 35 |
| 2.16. Image Processing Pipeline | 36 |
| 2.17. Filling the Image Processing Pipeline with Data | 37 |
| 2.18. Multiple Image Processing Pipeline | 38 |
| 2.19. Concept of the Parallel Plug-in for Integration | 42 |
| 2.20. Concept of a Parallel Plug-in for Image Processing | 45 |
| 2.21. Image Processing Pipeline with Acknowledgment Functionality | 46 |
| 2.22. Image Processing Pipeline with Service Plug-in | 46 |
| 2.23. Possible Ways of Loading Parallel Plug-ins | 49 |
| 3.1. Life-cycle of the Monte Carlo Integration Application | 58 |

| | |
|---|----|
| 3.2. Monte Carlo Integration with Additional Failure Handling | 59 |
| 3.3. Life-cycle of the Image Processing Pipeline Application | 61 |
| 3.4. The Harness Thread Pool and Parallel Plug-ins | 63 |
| 3.5. Thread Approach with Critical Section | 64 |
| 3.6. Image Processing Pipeline Implemented with Threads | 64 |
| 4.1. Architecture of the Parallel Plug-in Prototype Suite | 72 |
| 4.2. Design of the Prototype Suite Components | 74 |
| 4.3. Finite State Diagram for Integration Plug-in Component | 79 |
| 4.4. Nassi-Schneidermann Diagram for the Integration Algorithm | 80 |
| 4.5. Finite state diagram pipeline component | 81 |
| 4.6. Nassi-Schneidermann Diagrams for Image and Acknowledgment Passing Algorithms | 84 |
| 4.7. Finite State Diagram of the Parallel Plug-in Manager | 85 |
| 4.8. Nassi-Schneidermann Diagrams for Parallel Plug-in Loading Algo- rithms and Pipeline Restoration | 88 |
| 4.9. Nassi-Schneidermann Diagram for Integration Distribution and Redis- tribution Algorithms | 90 |

List of Tables

| | |
|--|-----|
| 2.1. CPU Time Dependent on the Dimension of the Integral [Has00] | 31 |
| 3.1. Tests for General Components | 67 |
| 3.2. Tests for PPM Components | 68 |
| 3.3. Tests for Integration Application Components | 69 |
| 3.4. Tests for Image Processing Pipeline Application Components | 69 |
| 4.1. Object IDs of the Exported Plug-ins | 76 |
| 4.2. Interface Definition of the PPM Plug-in | 76 |
| 4.3. Interface Definition of the Integration Plug-in | 77 |
| 4.4. Interface Definition of the Image Processing Pipeline Plug-in | 78 |
| A.1. Directory Structure of the Prototype Suite Implementation | 101 |

Abbreviations

| | |
|---------|--|
| DVM | Distributed Virtual Machine |
| HARNESS | Heterogeneous Adaptable Reconfigurable Networked Systems |
| HEC | High-end Computing |
| HPC | High Performance Computing |
| IP | Internet Protocol |
| IPC | Inter-Process Communication |
| MPI | Message Passing Interface |
| MPMD | Multiple Programs Multiple Data |
| NFS | Network File System |
| PPM | Parallel Plug-in Manager |
| PVM | Parallel Virtual Machine |
| RMI | Remote Method Invocation |
| RMIX | Remote Method Invocation Extension |
| RPC | Remote Procedure Call |
| RTE | Runtime Environment |
| SPMD | Single Program Multiple Data |
| TCP | Transmission Control Protocol |

1. Introduction

1.1. Overview

Today, parallel computing is used to solve large-scale problems. Parallel computing is the simultaneous execution of tasks on multiple processors. The aim is to receive results faster. The basic approach is to divide the problem into smaller pieces, which are coordinated and computed simultaneously. Parallel computing has become essential to the work of scientists and engineers around the world.

Research and industry use enormous amounts of computational performance for simulations and modelling in aerospace, medicine, nanotechnology and material science. High-End Computing (HEC) systems with the increasing number of processors and extensive growth of system scale have provided the means to new scientific breakthroughs. [fHEC]

There is also a shift from vector processors and massive-parallel-processor machines to the commodity-based clusters considerably cheaper to set up for the same level of computation power. Execution takes place mainly on High-Performance Computing (HPC) clusters, which are commonly used in scientific computing. Generally, clusters consist of compute nodes, storage capacity and an interconnection network. Many computing nodes are connected and transparent access to the resources is provided by software packages that allow the connected, distributed computers to be seen and used as one multiprocessor machine. [ES05][Chi][fHEC]

The terms High-End Computing and Supercomputing are also synonymous with High-Performance Computing. HPC can be defined as a branch of computer science which develops supercomputers, and software to run on supercomputers. An area of HPC is the development of parallel-processing algorithms and software, especially

programs, which can be divided into small pieces, so that each piece can be executed simultaneously. [otPotUS04]

Software organises the resources and also handles the communication of application components during the computation. Thus, many problems must be solved for the administration of distributed applications and needed communication.

Clusters are popular HPC implementations which run Linux as the OS and free software to implement parallelism. This configuration is often referred to as a Beowulf cluster. Beowulf is not a certain software name but is a reference to how the cluster is built up, usually with ordinary personal computers. Communication takes place via the Transmission Control Protocol / Internet Protocol (TCP/IP), a collection of protocols to connect computers from different vendors and to define the data exchange. [Beo05][Hol97]

A second important aspect of HPC is the development of appropriate parallel software, described in the next section, which offers information and data exchange. Many programs are specifically designed for writing sophisticated, scientific applications for HPC computers.

1.2. Previous Work

Two well-known parallel computing packages are the Parallel Virtual Machine (PVM) [GBD⁺94] and the Message Passing Interface (MPI) [SOHL⁺96]. Both packages deal with passing of messages and data between the different nodes running the application. Both solutions provide function sets for handling messages.

1.2.1. Parallel Virtual Machine

The PVM system is a research project addressing the heterogeneous network computing. The software package contains tools and libraries emulating an adaptable multiprocessor system on interconnected computers. These computers can be of varied architectures. The major objective is the usage of the participating computers for

parallel computations. [SL05]

PVM is based on several principles including

- User-configured host pool
- Transparent access to hardware
- Process-based computation
- Dynamic set of parallel tasks
- Explicit message-passing model
- Heterogeneity support

PVM tasks are independent, sequential threads of control responsible for communication as well as computation. There is also no process-to-processor match in PVM, thus multiple tasks can be executed on a single processor. The tasks are executed on a set of machines predefined by the user. The pool can consist of single-CPU machines as well as multiprocessor systems. The user application may exploit the capabilities of specific machines, or it may view the hardware as a collection of virtual processing elements. [GBD⁺94]

The message-passing model defines the cooperating of computational tasks by sending and receiving messages between processes. Message size is limited only by the amount of available memory. PVM supports heterogeneity in several ways. An important way is that messages may also be exchanged between machines having different data representations. [GBD⁺94]

PVM model is composed of two parts, a daemon process and a library. Before a user can run applications, the daemons which build the virtual machine must be started. The user can then execute one or several PVM applications from a UNIX prompt on any of the participating hosts. The library contains the interface routines. The routines comprise message passing, spawning processes, coordinating tasks, and modifying the virtual machine. [GBD⁺94]

1.2.2. Message Passing Interface

MPI is also a standard for message passing defined by the MPI-Forum. The MPI-Forum is a committee of vendors, implementers, and users. MPI is a library specification including a message-passing model for parallel computers, clusters, and heterogeneous networks. Like PVM, it is designed to enhance the development of high-performance scientific applications. [GL]

MPI has been strongly influenced by work from IBM, Intel and PVM. The main advantages of a message passing standard are portability, ease-of-use, and that vendors have the opportunity to work with a clearly defined set of routines. [Heb99]

MPI aimed to be a library for writing applications rather than a distributed operating system. It is also capable of delivering high-performance on high-performance systems. Like PVM, it is designed to support heterogeneous computing. Another goal is to accelerate the development of portable parallel libraries using MPI's modular approach. MPI is thus extensible to meet future needs. [Gro04]

1.2.3. Harness

Harness is a meta-computing system that provides a pluggable, heterogeneous Distributed Virtual Machine (DVM) environment. A DVM is a cooperating set of processes, which offers a wide range of services. Harness provides fault tolerance and organises programs and services by using plug-in software modules. The research goal is to improve application productivity and efficiency on dynamically changing structures and high-performance computing platforms. [Har05]

The project is a follow-on to PVM and an ongoing, collaborative work between the Oak Ridge National Laboratory (ORNL) [SL], University of Tennessee Knoxville (UTK), [oT] and Emory University [Uni]. The research partners have developed a variety of experiments and prototypes exploring a pluggable framework, highly available DVMs, and heterogeneous, reconfigurable communication frameworks. [FGB⁺04][Har05]

Harness is built on architectural features such as [BDF⁺98]

- A lightweight kernel consisting of a set of core functions, handling the basic components either locally or remotely. The components are implemented as a set of calls, processes and threads.
- The kernel can exist in a form of a daemon (a Harness application responding to requests from local applications) or remote daemons. Requirements to a daemon are message passing, to start or to stop processes or threads and the ability to start other kernels.
- A Harness DVM is a set of cooperating daemons. Together, these daemons provide basic features like communication or process control.
- Mechanisms providing possibilities for the dynamic management of system components.

1.3. Project Description and Objectives

One major part of the Harness software architecture is a runtime environment (RTE). This environment (kernel) is a flexible framework for plug-ins. The kernel provides basic features for the dynamic loading of plug-ins, which in turn provide a wide range of services such as communication capabilities, scientific algorithms and fault-tolerant applications. [EG05a]

A new concept in Harness is parallel plug-ins. These parallel plug-ins will assemble applications and provide services. Scientific applications will consist of several plug-ins. The field of parallel plug-ins offers many possibilities for fundamental research. Plug-in design for Harness development must be investigated.

The main objectives of this Master thesis project are

- Gaining experiences in the research area of parallel plug-ins.
- Using the available functions provided by Harness and extending Harness by developing prototypical parallel plug-ins that can be used as templates for many user applications.
- Exploring the new aspects of fault tolerance that arise from the use of dynamic parallel plug-ins.

A result of this Master thesis is a parallel plug-ins paradigm derived from experiences with this new technology. Investigations were based on the Harness workbench and existing parallel programming fundamentals. New ideas and existing principles were applied to parallel plug-ins adding new knowledge to the field.

An initial suite of plug-in prototypes were created to serve as paradigms for future modules and to prove the concept. I have investigated the potential of Harness functions and features, and considered their integration into scientific applications, their ability to operate such applications in the Harness environment. I have also considered the addition of basic fault tolerance to applications implemented as parallel plug-ins.

1.4. Requirements and Key Problems

A key feature of the Harness workbench is the use of pluggable modules. Depending on their functions, plug-ins must meet different requirements such as inter plug-in communication, plug-in loading, and fault tolerance.

Configurations of parallel plug-ins must be investigated. These configurations are connected to certain types of use cases and applications. Applications have different requirements on parallel plug-ins. Two areas open to future research are the dependencies and interactions among the components of a single parallel plug-in and the same considerations between separate parallel plug-in units.

1. Introduction

Interactions require communication, and there are different kinds of communication layers possible. A parallel plug-in can communicate

- With the kernel,
- Other parts of the parallel plug-in itself,
- Other local or remote plug-ins.

The interoperability and extensibility of parallel plug-ins is of vital importance. Thus, it is important to find different use cases of a parallel plug-in regarding its distribution on the machines and its communication capabilities. Communication is needed in different stages of the application. It starts with the coordinated loading of all the parallel plug-in parts and continues with the distribution of data and information.

Fault tolerance is another research area. Compute nodes can fail and parts of a parallel plug-in could become unreachable. Simple failure notification and handling mechanism are investigated. Failures must be identified and reported to other plug-in parts.

This Master thesis addresses the basic principles. The task covers the design and evaluation of parallel plug-in modules. General information and concepts regarding the development and operation of parallel plug-in applications are introduced and demonstrated. Possible implementations were derived from programming example applications. The resulting prototypes were expanded with additional capabilities such as simple failure handling. The prototypes and included functions may be used as templates for future scientific applications.

2. Preliminary System Design

2.1. Basic Principles and Concepts

2.1.1. Communication

The fundamentals of parallel applications are parallel algorithms. These algorithms describe data processing or problem solving. Algorithms are a set of instructions for performing a specific task. Starting from an initial state, the instructions are executed until end-state is reached.

Parallel algorithms take advantage of the availability of several processors. Thus, it is possible that a single problem might be processed simultaneously by many units. Often, a task is divided into subproblems, which are passed to the processing units. Results are collected after the parallel algorithm's instructions have been carried out.

Implementation of parallel algorithms is based on distributed data and available communication possibilities. Exchange of data and messages, are each of vital importance. Messages may include requests for executing provided functions, or information regarding failure detection and notification.

2.1.1.1. Inter-Process Communication

Inter-Process Communication (IPC) is a method for transmitting information between processes or programs. For instance, it is widely-used in multitasking operating systems. Furthermore, there are no restrictions regarding the location of the processes. They may exist on the same compute node, or on several compute nodes in a network.

IPC describes ways of transmitting messages containing information or data. Hence, the existence of many different use cases for IPC leads to the fact that there are a wide variety of IPC opportunities present as well. Today, most of the IPC systems are based on one of the following approaches.

- Shared memory
- Message passing
- Remote procedure call

IPC can be used for cooperative and competing processes. Cooperative processes exchange data explicitly, i.e. state information of one of the processes or computation results. Competing processes do not communicate directly with each other. These processes may compete for the same resources, such as memory space. Thus, they have to communicate indirectly to synchronise the access.

Shared Memory

Shared memory is a form of IPC, which can be used on shared memory computing systems. Within such a system, several processors access the same memory. An example is shown in figure 2.1.

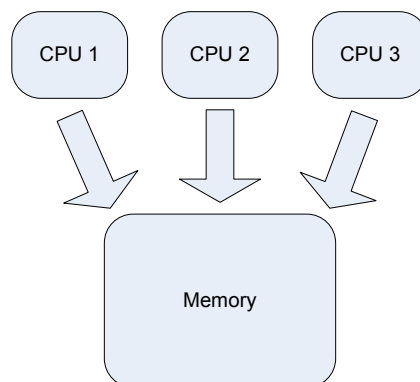


Figure 2.1.: Shared Memory System

The exchange of information is solved by allowing two or more processing units to address the same memory block. Figure 2.2 on page 10 shows the memory spaces of two processors, which overlap.

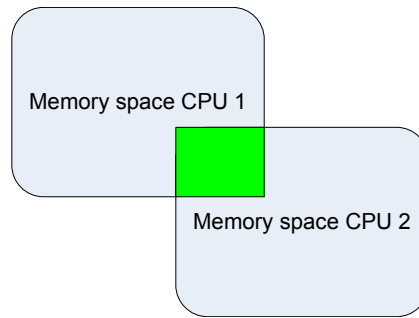


Figure 2.2.: Overlapping Memory Space

Problems regarding the consistency of data may occur when simultaneously accessing the same memory space with more than one process. For instance, two processes change the same variable. A section in memory accessed by several processes at the same time is called a critical section. This situation invokes the use of certain rules for access. The parallel execution of both processes must be sequential at this point.

One way to achieve this sequential access is a mutual exclusion, for instance, with semaphores. Semaphores can be seen as gatekeepers, allowing only one process at a time to enter the critical section. When the process has left the critical section, the next one can enter. [Her04]

Message Passing

Message passing IPC is often used in distributed memory systems. Each processing unit has its own memory space. For information exchange, the single units are connected via a network (see figure 2.3 page 11).

The message passing system is reliable for the transportation of data over the network connection. This can be compared to a mail system. Each processing unit has a mail box. If one process wants to exchange information with another process, the data is copied, and by calling a send directive, it will be sent to the addressed mail box. By calling a receive directive, the other process can receive data from its mailbox.

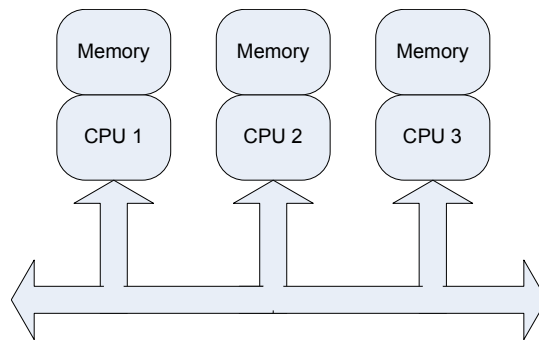


Figure 2.3.: Distributed Memory System

Remote Procedure Calls

Remote procedure calls (RPC) are based on the invocation of functions or services provided by another process or server. Thus, RPCs are mainly used in client/server applications (see page 12). In contrast to shared memory or distributed memory, the client and the server processes do not have to run parallel in time.

The server waits for calling clients, processes their requests, returns the response and waits again. If there are no requests to process, the server remains in an inactive state. Furthermore, there exist different forms of RPCs. The classical implementation is a synchronous RPC. Further development led to the asynchronous RPC.

Synchronous RPCs are handled like a local function call. The application starts an invocation, which is processed by a stub on the client side. This client stub takes the parameters, packs them, and sends them over a network connection to the server. On the server side is a so-called server stub, which receives the method invocation and unpacks possible parameters. Then, the server-side function is called and performed. Possible return values are sent back by using server and client stubs in reverse order. The client process is blocked while sending the request and waiting for an answer.

Asynchronous RPCs are similar to synchronous RPCs, but there is one main difference. The RPC function is called by the client application, but the program does not wait for an answer from the server, and the function call returns immediately. The main advantage is the ability of the client to perform further tasks while the request

is processed by the server application. After the server finished the request, the answer can be picked up by the client application.

A third kind of RPC implementation is also imaginable. The RPC call may be performed one way only. A remote function is called, i.e. to invoke a certain instruction sequence on the remote side. Unlike the two other types of RPCs, no return value is expected. Hence, this implementation of RPC annuls, in a particular way, the client/server principle. A process is not requesting a service from another process, but initialises it to perform a certain action.

2.1.1.2. Communication Concepts and Topologies

Communication Concepts

Two main concepts or architectures can be distinguished. On the one side, there is the client/server approach, and on the other side the peer-to-peer approach. The main difference between these concepts can be traced back to the role played by each of the participating elements.

A client/server system consists of a participant, acting as a client, opening a connection to another process or program which acts as a server. Often, the client offers a kind of user interface of an application. The server provides functionalities. This model emerged at a time when computers filled entire rooms. Users could access the computer via terminals [Sch03].

Within a client server model, the server supplies one or more clients with certain services. It accepts all requests and organises their processing. Typical services of the client/server architectures are authentication in networks, printing services, or the solving of special computations which require defined amounts of computation power or special libraries.

Concerning applications like database requests or mathematical computations, the server may be equipped with appropriate computing power. At this point, one of the possible disadvantages must be mentioned. In particular cases the scalability of the client/server model cannot be fulfilled. Usually, a server can only perform a certain

number of requests. When the amount is reached, there is a bottleneck at the server.

Another disadvantage is the single point of failure. If the server fails, the whole client/server application fails, and requests cannot be processed. One way to overcome this issue could be replication of the server.

Generally, the server address and its interface must be known by client processes in order to access it. Interface means the services the server can provide, and the methods necessary to invoke these services. Two elements form the interaction: request and response (figure 2.4).

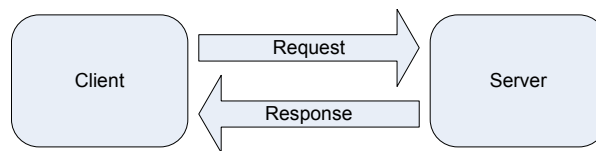


Figure 2.4.: Client/Server Model

The second communication concept is a peer-to-peer model, which is a distributed system. Each member of the system is simultaneously client and server. The peer uses and provides services. Peer-to-peer networks are decentralised without coordinating or naming and location authorities.

Typically, the nodes are connected via largely ad-hoc connections. Main uses are the sharing of content files, (audio, video, data or anything in digital format) and the passing of real-time data, such as telephony traffic. This type of communication differs from that of the client/server model where the communication takes place between the clients and the centralised server. [MKL⁺02]

Highlighted advantages are scalability, which is based on the constant joining and leaving of peers, and the self-organisation of the network, as well as the elimination of a single point of failure. But some problems also exist, for example, the missing of a central control and the weakness of security and availability. Peer-to-peer networks also have a bad reputation due to file-sharing platforms. [MKL⁺02]

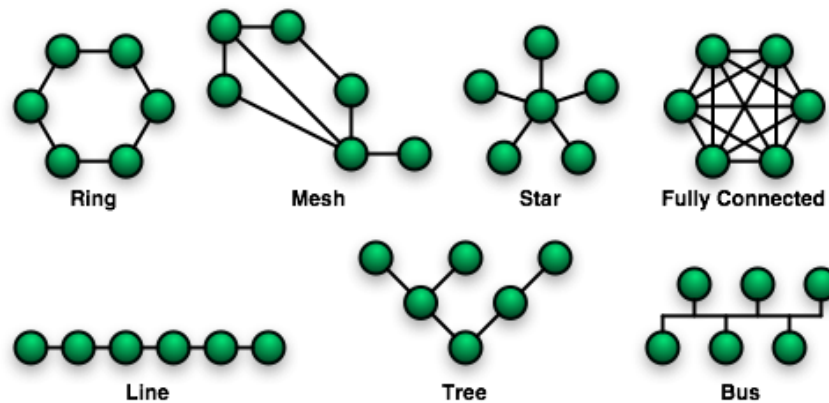


Figure 2.5.: Network Topologies [Wik06c]

Communication Topologies

Topologies address the areal distribution of communication systems. It describes how nodes are linked to others, and a great variety of linking possibilities exist. Examples are presented in figure 2.5. The topology only determines the configuration of connections between nodes. [Sch03]

Topologies may be partitioned into chains, centralised and decentralised patterns. A chain system is bus-based, and a node can easily be integrated by connecting it at the end of the existing series of computers. A message, sent in the network, passes all participants. Two pictured chain topologies are bus and ring. A disadvantage of the chain topology is the fact that if a single link fails, the entire network may break down. [Sch03]

A second possibility is to base the topology on centralisation. The star topology, for instance, reduces the chance of network failure as all nodes are connected to a central node. The failure of a transmission line isolates the peripheral node, which is connected via this line to the central node, but all the remaining nodes will be unaffected. [Sch03]

The tree is also a centralised topology and occurs like a collection of connected star networks. If a link to an end node (leaf) fails, the leaf is isolated as in a star topology. But if a connection to a non-leaf node fails, an entire section becomes unreachable. Advantages of the tree are: time costs to reach the lowest level, and security via en-

capsulation. [Sch03]

Mesh and fully connected graphs are decentralised topologies. In a mesh, at least two nodes have two or more links between them. In a fully connected graph, each node is connected to all the other nodes. The main advantage is high reliability. If one node fails, all the other nodes can still communicate with each other. In a fully connected topology with n nodes, there are $n(n-1)/2$ links. A great disadvantage of this topology is the high cost. [Sch03]

Criteria by which to evaluate a topology are availability, connectivity, scalability, bandwidth, set-up cost, failure and security. Furthermore, it is possible to connect simple topologies to create a new one, for instance, the tree consisting of single stars. It is also possible to overlay a topology with another logical one. For instance, within bus or fully connected topologies, it is possible to create a ring by always accessing particular nodes in a certain sequence. [Sch03]

2.1.2. Harness

Today, three different Harness prototypes exist, two C variants and a Java-based implementation. Each of them is concentrated on different research issues. The provided Harness runtime environment concept has two main parts, the kernel including core functions, and a set of plug-in software modules. These modules may provide services such as messaging or computational algorithms. [EG05a]

The kernel is a container and provides functions for loading software components. The Harness kernel consists of three main parts (see figure 2.6 on page 16). An external daemon process starts up the three components of the kernel. These three components are a process manager, a thread pool and a plug-in loader. [EG05a]

The process manager offers access to the standard input/output of child processes and provides facilities for the creation of child processes. This is a necessary element for the execution of external programs, e.g., a shell. [EG05a]

The thread pool offers functions which simplify the use of threads. The kernel and the execution of jobs are based on a thread concept. Jobs are submitted to the thread

2. Preliminary System Design

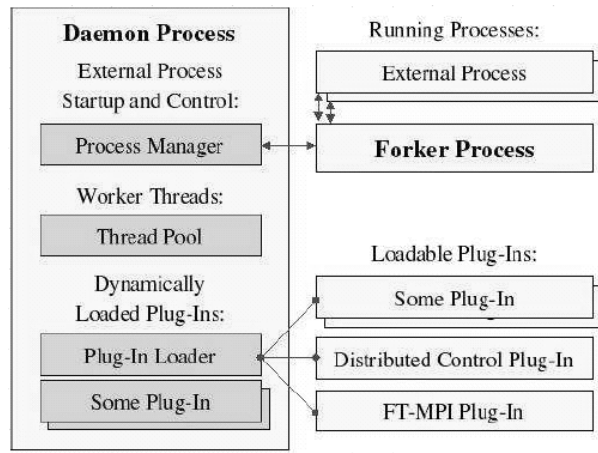


Figure 2.6.: Lightweight Kernel Design [EG05a]

pool and executed, or if necessary, queued. [EG05a]

The third component is the plug-in manager. The manager is an internal handler of the kernel performing the loading and unloading of plug-ins. The manager is responsible for embedding the plug-ins into the kernel environment. Embedding into the Harness kernel grants the plug-in access to the kernel functions, which allows it to execute threads and external processes. [EG05a]

Plug-ins are implemented as shared libraries. They can be loaded by the user during the start of the kernel, and is also possible for one plug-in to reload other plug-ins. Additional plug-ins are loaded if necessary and the system is reconfigurable. [EG05a]

Harness is being continuously improved. Today's fields of research are different types of plug-in applications, the concept of parallel plug-ins itself, as well as a reconfigurable communication framework, and the merging and splitting of multiple virtual machines.

2.1.3. Remote Method Invocation Extension

The Remote Method Invocation Extension (RMIX) is a dynamic, heterogeneous, reconfigurable communication framework. It is capable of using various Remote Method Invocation (RMI) and RPC protocols. The protocols can be exchanged by

2. Preliminary System Design

dynamically loadable plug-ins providing different protocol stacks. [EG05b]

The RMIX framework was currently integrated into Harness. This C variant of RMIX is based on the heterogeneous and reconfigurable communication framework originally developed at Emory University in Java. Functions are provided for communication via TCP/IP where RMI and RPC protocols can be integrated, e.g., Sun RPC, Java RMI or SOAP. The RMIX framework allows access to remote network services. [EG05b]

RMI is an important communication paradigm for heterogeneous distributed environments. Local method calls are transferred into network systems. Clients can invoke methods provided by a local or remote server. For the invocation, a protocol stack must be used to define connection management, message formats and data encoding. [EG05b]

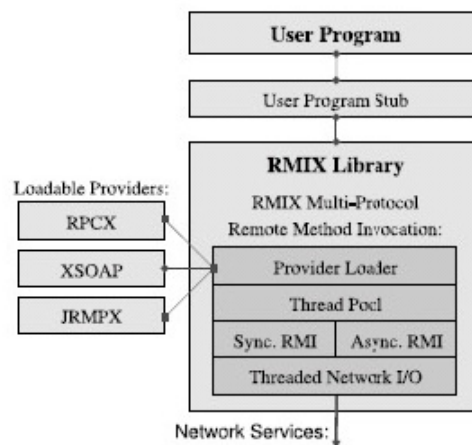


Figure 2.7.: RMIX Framework Architecture [EG05b]

Figure 2.7 shows the architecture of RMIX. It consists of two parts, a library and loadable providers. The library includes functions used by all protocol stacks, such as networking and thread management. The provider modules contain protocol specific functions, for instance, connection management or message formats. [EG05b]

Furthermore, the library reuses software developed for the Harness RTE, the thread pool for handling and managing the threads, as well as the module technology for the providers to allow dynamically protocol changes. [EG05b]

2. Preliminary System Design

The C variant of RMIX supports synchronous and asynchronous RPC calls. RMI calls are mapped to RPC calls in the base library. Therefore, an object registry is used to store object interface information on the server. When a server exports an object, method names, signatures and respective stub function pointers must be defined by the user. This information characterises the object interface. [EG05b]

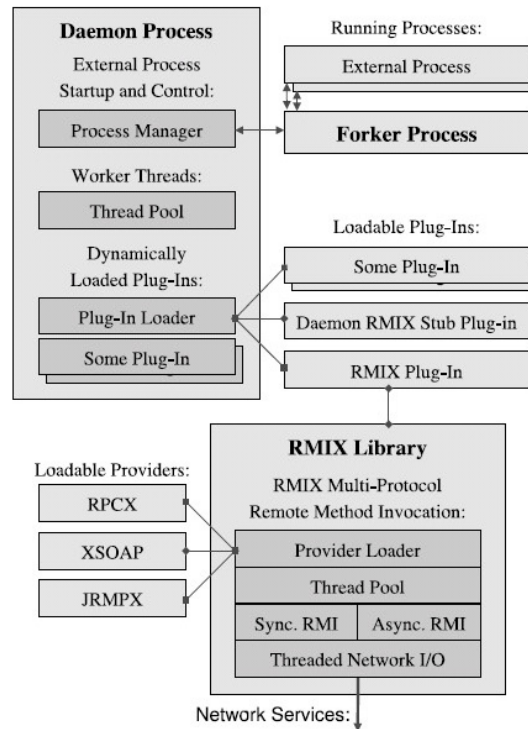


Figure 2.8.: RMIX Plug-in for Harness [EG05b]

Incoming RPC calls are handled by server-side object stubs and transformed to RMI calls. On the invocation side, client stubs adapt outgoing RMI calls to RPC calls. The stubs are also responsible for the conversion and packing of possible parameters sent via the network connection. [EG05b]

The C-based RMIX variant is integrated into the Harness RTE in form of a plug-in (figure 2.8) to provide RMI/RPC capabilities to the RTE and to other plug-ins. The RMIX base library and Harness RTE stubs are implemented in a Harness-RMIX plug-in. Third party developers need to implemented stub functionalities into their plug-ins as well. If a plug-in wants to use RMI/RPC calls it must load its stub plug-in(s),

which will automatically load the RMIX plug-in. [EG05b]

2.1.4. Fault Tolerant Design

The aim of fault tolerant design is to continue the operation of a system after a failure occurs. Therefore, design methods must be applied to continue more or less fully operational. A reduced level of functionality may be possible, for instance, a reduction in throughput or response time. This may happen after a partial failure: parts of the system break down, but the others continue to function. [Wik06a]

For implementing fault tolerance, fault-tolerant components can be used to continue the work even when a subsystem breaks down. Another possibility is the use of redundancy. Here, backup system components will take over the task of a failed component. Redundancy uses the technique of duplication where different approaches are possible. [Wik06b]

- Replication provides multiple identical instances of the same system. Requests are directed to all of them in parallel and the correct result is chosen on the basis of a quorum.
- Redundancy provides multiple identical instances of the same system. In case of a failure, the system switches to one of the remaining instances.
- Diversity provides multiple, different implementations of the same specification. These implementations are used like the replicated system to cope with errors in a specific implementation.

Fault-tolerant design is often used in combination with a fault-detection system. Otherwise, failures may not be detected by the user or operator. The failure must be corrected. For example, running on a backup system for a prolonged time period is not a solution, as the backup could also break down. Thus, it is of vital importance that crashed components will be repaired.

Two possible forms of recovery in fault-tolerant systems are roll-forward and roll-back. In the case of the roll-forward approach, the system state is taken at the time

the error occurs and correction is attempted. After the correction, the system or application can move forward from the repaired point. Roll-back recovers the system by moving it to a previous state, which is known to be correct. An example is checkpointing. Checkpointing creates an image of the system state and stores it for later recovery purposes. [Wik06b]

The design of fault tolerant systems also raise new problems, for example, regarding the test techniques of such systems. First, it is impossible to consider all possible faults which might emerge in a system, and second, some test cases cannot be implemented in reality. The proper functioning of a backup system cannot be guaranteed. An examples would be the backup system of a nuclear reactor. [Wik06a]

Cost is another problem. The integration of fault-tolerant components or redundant systems may often dramatically increase cost. More hardware may be needed, or in the case of software development, more quality management and programming hours. The cost can be economic, or physical, such as weight or dimensions. For example, the use of backup computers often requires more space and if space is restricted, priorities must be set regarding needed components and back-up components. Furthermore, it is unnecessary to provide every component with fault tolerance or a backup system. Decisions must be made concerning the importance of a system and how critical it is. Possible criteria could be the failure rate of a certain component or the economical use of it. [Wik06a]

2.1.5. Today's Plug-in Technology

Plug-ins are programs interacting with other programs to provide them certain, specific functions. A plug-in extends software, such as a browser, to provide new features. Therefore, a plug-in must interact with its backbone software. Plug-ins are usually integrated over a well-defined interface. For example, the handling of special data types not included in the basic software, such as playing multimedia files, encrypting and decrypting email or filtering images. [Hei05]

Examples of programs extendable by plug-ins are Netscape, Mozilla Firefox and Photoshop. Mozilla Firefox may be amended by plug-ins from different vendors or soft-

2. Preliminary System Design

ware developers to add new capabilities, i.e. opening of new file formats such as special document files (Adobe Reader) or multimedia files (QuickTime).

Plug-ins are also a possible way to overcome bulky software. To satisfy user's demands for new gimmicks, many software developers include all available features in their systems. These features are often loaded every time the program is started and executed, which slows the program and the system. Plug-ins allow the dynamic addition of features during runtime. [MMS02]

A plug-in is mostly unknown at the compile time of the application. Therefore, no information concerning a special plug-in or several plug-ins are mentioned in the source code of the original application. Thus, the process of dynamic loading has special requirements, and the plug-ins themselves require the application for which they were designed. As they access the special interface of the application, they cannot be loaded into another one. [MMS02]

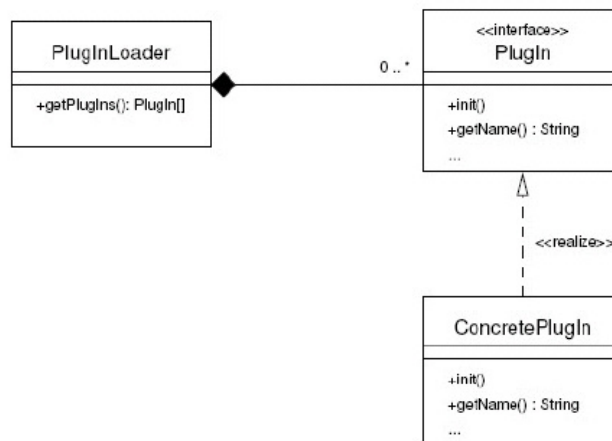


Figure 2.9.: UML Class Diagram for a Plug-in Pattern [MMS02]

Figure 2.9 presents a class diagram showing a plug-in pattern. The picture illustrates classes and interfaces. The **PluginLoader** is responsible for searching plug-in interfaces at runtime. This takes place when a certain plug-in interface is needed. The loader may also be responsible for invoking plug-ins for initialisation purposes. Another task performed by the loader is to grant access to all loaded plug-ins. [MMS02]

The second part of the pattern is the PlugIn Interface. An interface is provided for communication with a certain plug-in type. It may include different method, i.e., getName() to access the name of the loaded plug-in. The third part is the ConcretePlugIn, which implements the interface and provides special features. [MMS02]

The PlugInLoader reveals the names of the plug-in by using the getName() function call, and is responsible for initialisation via the init() function of the plug-in. Then, plug-in methods can be accessed by using the interface functions of the plug-in, as well as functions of the plug-in loader. [MMS02]

This pattern is portable to the concept, which is used by the plug-in loader integrated in the Harness runtime environment.

2.2. Definition of Parallel Plug-ins

This Master thesis project intended to merge today's available techniques and technologies such as the Harness runtime environment with the plug-in pattern to create a new concept of parallel plug-ins.

Harness is a pluggable, heterogeneous, distributed virtual-machine environment for parallel and distributed scientific computing. Like PVM and MPI, Harness offers features that allow programmers and scientists to create parallel applications which run on several machines in order to access a great amount of computing power, and to exploit their available compute power.

One main goal and motivation factor of Harness is the enhancement of the overall productivity of scientific applications on diverse HPC platforms. The Harness project also aims to optimise development and deployment processes and to facilitate software reuse. Applications running on Harness including their basic designs must be adapted to new requirements as well. The advantages of Harness must not be wasted by bulky and overly complex software and applications. The plug-in approach offers new possibilities for design applications, whose components are only loaded when they are needed. This yields a clean application.

Plug-in terminology, combined with the access to parallel and distributed HPC plat-

forms provided by Harness, promise the development of a new design concept for software: the parallel plug-in concept. This new design attempts to use the advantages of both technologies to provide scientists easier access to available computing power as well as enable programmers to reuse already developed code. This increases the efficiency of the overall software development process in the world of parallel computing.

Compared to already known plug-in technology, parallel plug-ins move one step further. Parallel plug-ins not only amend applications on distributed machines, but also build up whole, modularised, pluggable applications running on a distributed virtual machine.

2.3. Motivation and Features of Parallel Plug-ins

The motivation of parallel plug-ins is the transfer of common elements from the field of architecture to the software engineering sector. In architectures, the reuse of common designs for solving different problems is widely used. Now, certain software design concepts can be reused and pattern can be developed.

These patterns and guidelines for parallel plug-ins also offer less-experienced users and programmers the ability to reuse, i.e., existing code or parallel plug-in frames, to adapt them for their own special needs. Design patterns are easy to understand and they offer practice-based solutions for existing design problems.

This Master thesis provides programmers and scientists with a design pattern for parallel plug-ins, as well as basic parallel plug-in frames illustrating different approaches to the new technology. The parallel plug-in concept builds on the extension of the Harness kernel with application features during the runtime.

Furthermore, it facilitates the modularisation of huge and complex software systems. This is a very important feature, as it reduces the overall complexity of scientific applications. Functionalities are separated over several plug-ins, loaded if necessary, and reusable by loading into other applications. Like libraries, existing application functions need not be written again and again.

Another feature is independence from other modules of the complete system. Parallel plug-ins or their components can be modified without influencing or rewriting other software parts. Only the access to the plug-in, the interface, must not be changed, as it defines access to the provided features of the parallel plug-in module.

Other users are also able to reuse the code or parallel plug-in parts without knowing the internals. The interface for accessing the module is the only needed part of interest to third party users.

Parallel plug-in technology offers flexibility for the Harness environment and its users. The environment can execute or load several parallel plug-ins, running simultaneously. Now, parts of a parallel plug-in or a whole parallel plug-in may be unloaded without affecting the Harness environment or other running applications. Plug-in technology offers another important asset in that long-running servers or applications which cannot be restarted, are not stopped, or even influenced.

2.4. Types of Parallel Plug-ins

A parallel plug-in extends the Harness workbench with new features and capabilities. It can provide functions usable in other parts of the Harness framework, but it can also be a single application spread over various nodes. A Harness parallel plug-in is an application or function library, which can be distributed or replicated over a virtual machine.

Depending on the purpose of possible applications, it is possible to derive the demand for different architecture concepts for parallel plug-ins. On one side, there are large, scalable problems which can be solved by dividing them into parts and always performing the same computations on each of these parts.

On the other side, there are also problems which are solved by applying different computations on the data, and where the computations can be parallelised. Figure 2.10 on page 25 presents different architectures of parallel plug-ins. Five compute nodes are illustrated, representing a distributed machine. On these compute nodes, different configurations of parallel plug-ins are running.

2. Preliminary System Design

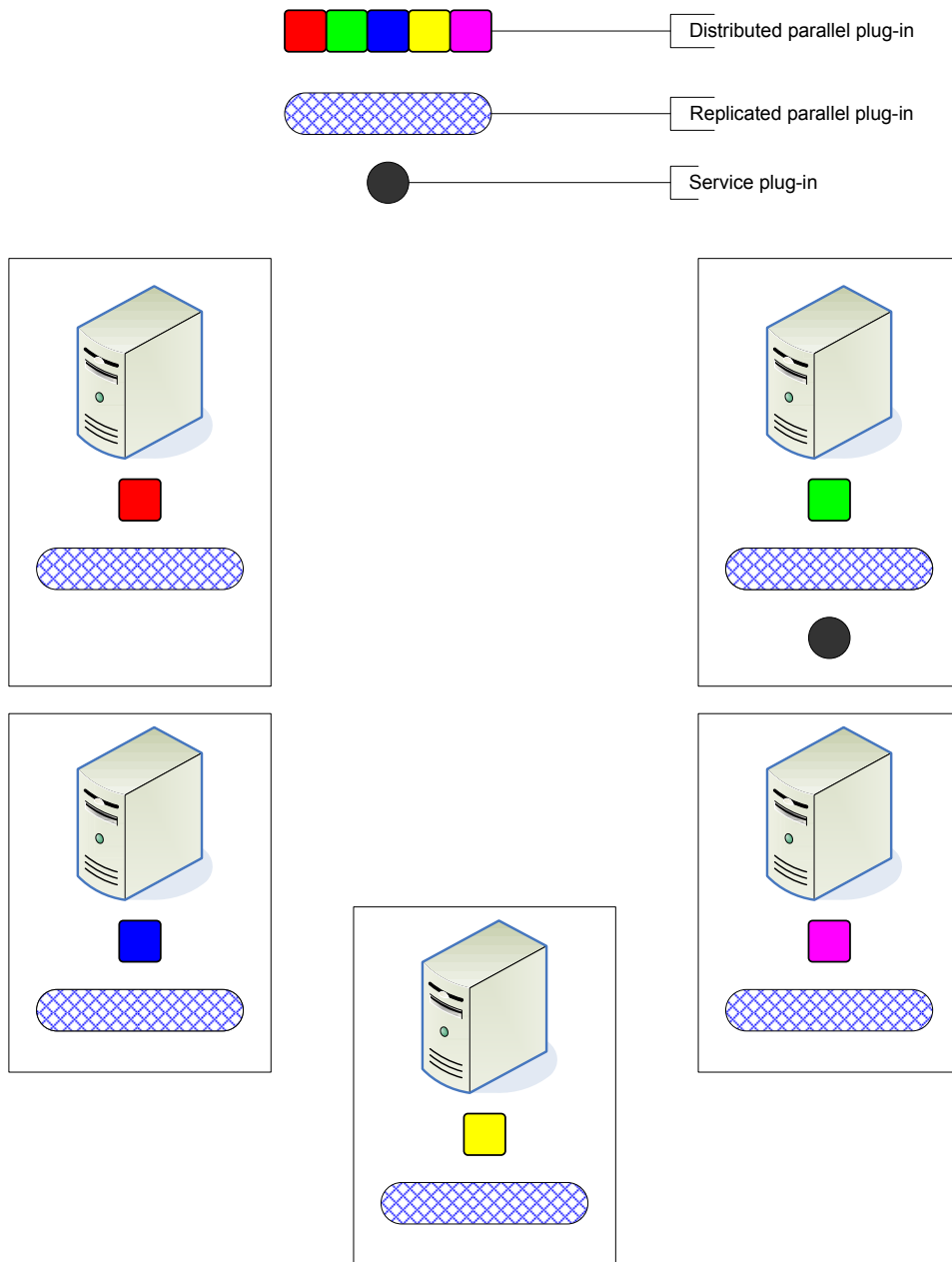


Figure 2.10.: Types of Parallel Plug-ins

2.4.1. Distributed Parallel Plug-in

The first parallel plug-in architecture is the distributed one. The parallel plug-in is a composition of plug-ins, which may work on separate nodes. These plug-ins may perform different tasks and need not be of the same kind. All parts must work together to achieve the expected results.

In figure 2.10 page 25 the distributed parallel plug-in is represented by a composition of several squares in different colours. Each colour represents one module of the whole parallel plug-in.

In this case, one can see that all units of the parallel plug-in are distributed over the available nodes. In the illustration, they are also equally distributed. It is also possible that two or more units of one distributed parallel plug-in run on the same compute node as Harness, or especially the kernel plug-in loader is able to load several plug-ins on one node.

The goal of the distributed parallel plug-in is the parallelisation of different tasks distributed on several machines. Each plug-in unit has a special kind of work to perform. Thus, the units are mostly a collection of different libraries and each has its own source code and its own duties to fulfil.

Since all the units must work together to solve the problem, communication between each of the parallel plug-in units is very important. Connections must be established to transmit information and data among them.

Dependent on the task to be performed, special communication structures may be built up. Possible structures are bus, ring or even a tree. Furthermore, this kind of parallel plug-in is more suitable for peer-to-peer architectures, as each unit performs its special instructions on the data and sends the result to the next, preset plug-in unit.

Thus, each unit that has a simultaneously active and a passive role may be regarded as a peer. To solve the main problem, a unit offers special services, in the manner of a server, to process data in a certain manner, but it also has a client role by requesting the service of another plug-in unit and forwarding the data to it.

2.4.2. Replicated Parallel Plug-in

The second variant is a replicated parallel plug-in. On each node, the same plug-in is replicated and loaded. This can be used for calculation problems, where each plug-in provides the same functionality on different data, e.g., the computation of a mathematical problem divided into separate parts.

In figure 2.10 page 25 the replicated parallel plug-in is represented by a bar. The parallel plug-in is loaded on all available compute nodes. Therefore, it is possible to speak of a replication of a plug-in, which builds up a new parallel plug-in.

In the case of the replicated parallel plug-in, it may also be possible for two or more plug-in units to run on the same compute node because of capabilities of the Harness plug-in loader. A disadvantage of running several plug-ins on one compute node is that they must share processing power.

All the units of a replicated parallel plug-in have the same architecture and they are internally the same, built on the identical source code, and providing the same services. Each unit can only perform tasks which can be executed by all the others.

This kind of a parallel plug-in is a good choice for problems which can be divided into smaller subproblems, each solved by applying the same algorithm. Therefore, communication between the units of a replicated parallel plug-in is not always necessary, unless certain computations performed by one unit are dependent on results from another plug-in. But that is dependent on the applied algorithm.

Furthermore, the failure of one plug-in unit is not tantamount to the cancellation of the whole task, as the subproblem may be recomputed by another plug-in unit.

In general, the units of a replicated plug-in are more independent from each other than the units of a distributed parallel plug-in. A possible application topology may be a star. All the units work as servers offering services, and another plug-in performs requests and uses the provided services.

2.4.3. Service Plug-in

The third kind of plug-in is a service plug-in. It is loaded locally and supports other loaded parallel plug-ins. For example, service plug-ins may provide coordination or scheduling.

A service plug-in may be described as a little helper for a parallel plug-in. Possible uses for service plug-ins are the provision of additional features for solving problems or the amendment of particular communication protocols, or even the handling of parallel plug-ins.

Depending on the purpose of the service plug-in, it can be loaded only once on a Harness kernel and offers its services locally to all the plug-in units loaded on that kernel, too. On the other hand, it might also provide remote services for all other plug-ins running on different machines.

When adding new communication facilities, it is possible that the service plug-in could be locally loaded on all available Harness kernels. In all cases, the service plug-ins are independent and need not communicate directly with each other. They provide functions for the support of parallel plug-ins. The task execution of parallel plug-ins will be simplified.

One example is the Harness RMIX plug-in. It is loaded locally on each Harness kernel, and provides the capabilities so that the Harness kernel and the loaded parallel plug-ins can communicate with each other.

Another possibility is a plug-in which handles an entire parallel plug-in. Parallel plug-ins perform tasks, but they must first be loaded, or after finishing, unloaded.

Figure 2.11 on page 29 shows different ways of loading parallel plug-ins. One option uses a service plug-in. The user may also load a parallel plug-in, independent from the type of parallel plug-in, on each compute node by him- or herself. It might also be possible to load only a service plug-in on one of the nodes.

The service plug-in can handle the loading and possible initialisation of all the plug-in units belonging to the parallel plug-in. The final state may be one or more loaded and initialised parallel plug-ins, which are ready to perform parallel applications. Besides

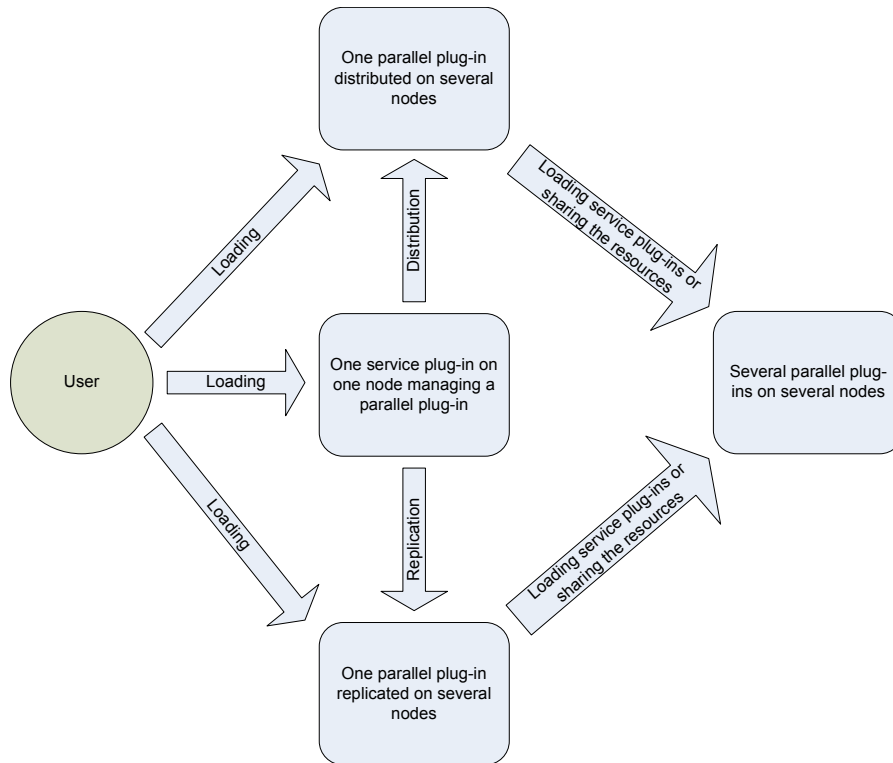


Figure 2.11.: Loading of Parallel Plug-ins

the loading of the parallel plug-in, the service plug-in may also be responsible for its unloading after the completion of the task.

2.5. Scientific Applications and Parallel Plug-ins

Mathematics, numerical analysis, data processing and visualisation, image processing and databases, are some of the fields where parallel computing is playing a decisive role. Much of today's human knowledge is based on computer simulations, i.e., bioinformatics, including genome research, testing of chemical structures or earthquake effects. These cases are performed as simulations because many of them cannot be tested in realistic environments. All these examples share a need for an enormous amount of computing power. But beside hardware needs, the implementation of applications and simulations exploiting the benefits of parallel and distributed systems are also of vital importance.

Harness, in combination with the concept of parallel plug-ins, attempts to build a basis for implementations of such scientific applications. Like PVM and MPI, Harness and the Harness RMIX plug-in offer access to the hardware and include interprocess communication, while the user does not have to be directly aware of the special types of hardware.

Now, the new concept of parallel plug-ins advises scientists and programmers to implement their applications with the benefits of this new pattern like code reuse and lightweight software. Two use cases were chosen as examples for possible parallel plug-in realisations. With the help of these examples, the opportunities provided by parallel plug-ins were researched and immediately applied in a practical manner.

2.5.1. Monte Carlo Integration

Many simulations in physics or other scientific disciplines are often based on the computation of integrals. The numerical integration of a function $f(x)$ and the boundaries t_0 to t_1 is

$$\int_{t_0}^{t_1} f(x)dx = \lim_{n \rightarrow \infty} \frac{t_1 - t_0}{n} \sum_{i=1}^n f(x_i) \quad (2.1)$$

where x_i are supporting points.

$$x_i = t_0 + \frac{i}{n}(t_1 - t_0) \quad (2.2)$$

Real functions can often be solved without a great effort. Integrations with higher dimensions are more sophisticated (see equation 2.3).

$$I = \int dx_1 \int dx_2 \cdots \int dx_D f(x_1, x_2, \cdots, x_D) \quad (2.3)$$

For instance, if the integration boundaries are zero to one for each dimension and the distance a between two supporting points is also equal in each dimension, then the number of supporting points can be expressed with equation 2.4

2. Preliminary System Design

$$n = \left(\frac{1}{a}\right)^D \quad (2.4)$$

If $a = 0.1$ the number of supporting points is

$$n = \left(\frac{1}{a}\right)^D = \left(\frac{1}{0.1}\right)^D = 10^D \quad (2.5)$$

For time measurements, the calculation time of one supporting point is assumed with $10^{-7}s$. The table 2.1 shows computation times for different integral dimensions. The demand for other solution approaches arises.

| Dimension | CPU time |
|-----------|------------|
| 1 | $10^{-6}s$ |
| 2 | $10^{-5}s$ |
| 3 | $10^{-4}s$ |
| 4 | $10^{-3}s$ |
| ... | ... |
| 10 | $10.7min$ |
| 11 | $2.8h$ |
| 12 | $1.16days$ |

Table 2.1.: CPU Time Dependent on the Dimension of the Integral [Has00]

Monte Carlo methods are an important utility in natural and engineering science, for instance in physics. These methods are used to solve complex integrations. Monte Carlo methods are based on numerically generated pseudo random numbers. These random numbers have special properties like a great period, homogeneous distribution and no correlation. For the generation of the random numbers mathematical libraries are used. [MR03]

An example, where random numbers occur, is dice. Each of the six numbers can be rolled with the same probability and the event is not predictable. As it is not practicable to link a physical system, like dices or radioactive decomposition, to a computer, pseudo random numbers are used which are often generated by applying simple rules. [Has00]

One important characteristic of a random number generator is the period, which indicates when a certain number is repeated in a series. A good generator has a great period. Furthermore, the distribution of random numbers plays a role. Random numbers can be homogeneous or inhomogeneous distributed. With the help of mathematical functions the distribution of random numbers, created by a particular generator, can be changed. [Has00]

The usage of Monte Carlo is supported by the availability of many compute nodes. So, the application runs as a distributed application. Generally, Monte Carlo functions simplify the calculation of complex integrals and integrals with complex boundaries. The utilisation of Monte Carlo is widespread. [MR03]

The Monte Carlo Integration is chosen as a use case, because the integration of a function with certain boundaries can be segmented in independent parts (see figure 2.12). Therefore, the Monte Carlo method can be realised as a replicated parallel plug-in.

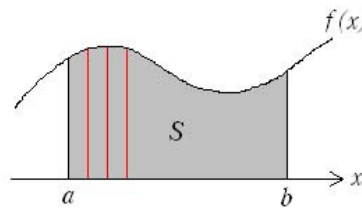


Figure 2.12.: Integral Divided into Chunks

During the project, the use of Monte Carlo was subject to restrictions. The main focus was on the investigation of parallel plug-ins. Therefore, only a function for the computation of a one dimensional function $g(x)$ was implemented. The integral of such a one dimensional function can be written as the equation 2.6.

$$I = \int_a^b g(x)dx \quad (2.6)$$

The integral can be expressed by an expected value $E[g(x)]$ of a homogeneous probability density function $f(x)$ in the interval $[a, b]$. [MR03]

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{other} \end{cases} \quad (2.7)$$

$$E[g(x)] = \int_{-\infty}^{+\infty} f(x) \cdot g(x) dx = \frac{1}{b-a} \int_a^b 1 \cdot g(x) dx \quad (2.8)$$

$$\int_a^b g(x) dx = (b-a) E[g(x)] \quad (2.9)$$

The expected value is approximately calculated with equation 2.10, a sum of n uniformly distributed random numbers in the interval $[a, b]$. [MR03]

$$E[g(x)] \approx \frac{1}{n} \sum_{i=1}^n g(x_i) \quad (2.10)$$

From this follows that the original integral I is roughly computed by the Monte Carlo Integration via equation 2.11

$$I \approx I_{MC} = \frac{b-a}{n} \sum_{i=1}^n g(x_i) \quad (2.11)$$

The Monte Carlo Integration is a suitable example as it is able to highlight the characteristics of parallel plug-ins. As the integration is a mathematical tool used in many applications and simulations, a parallel plug-in implementation of the Monte Carlo Integration may be reused by other scientific programs which additionally load the Monte Carlo plug-in. The access to the integration methods is allowed by well-defined interfaces.

At a later date, the parallel plug-in may be extended for providing different integration approaches. Due to the pluggable approach, the integration functionalities have only to be loaded if necessary and can be unloaded after the computation. So, resources can be reused for other purposes.

2.5.2. Image Processing Pipeline

Image processing is a computation problem which needs on the one side a lot of processing power for the calculation of certain filter functions and on the other side memory for storing the images during the computation process. Mostly, images are processed with a variation of filters. Appropriate functions can be rotation, cropping or the use of various filters, for instance an oil painting filter.

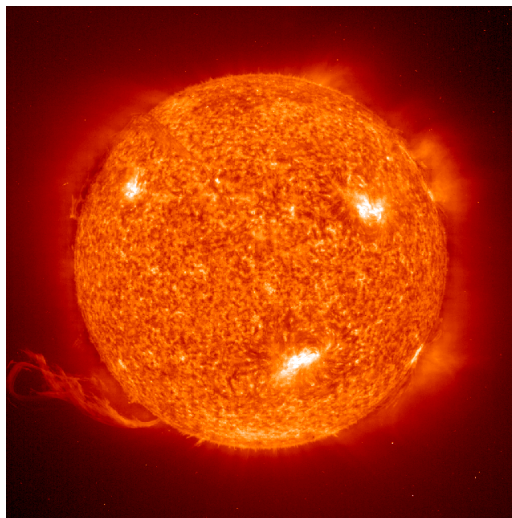


Figure 2.13.: Source Image

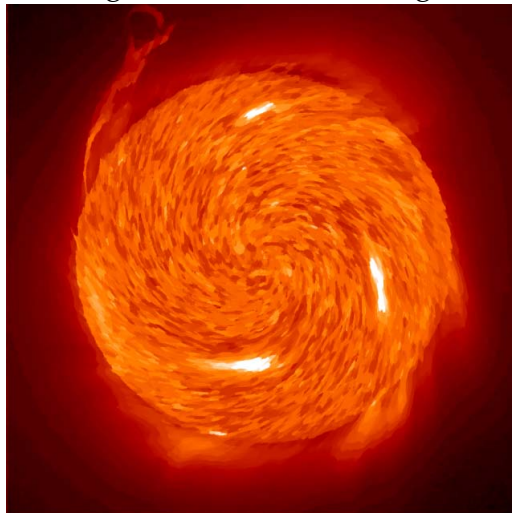


Figure 2.14.: Processed Image

The editing of an image is a serial process. To achieve a certain result, the image has

to pass several filters one after another. The figures 2.13 and 2.14 on page 34 show an example for a processed image. The original pictures was edited by using an oil painting filter, rotating it by 90 degrees to the right and swirling the centre of the image.

Two approaches exist to parallelise the problem. One option is to have several instances, which perform the filters on each image. In this case, the instances are independent from each other and fully equipped with all the needed functionalities (figure 2.15). With three units, it is possible to process 3 images at the same time.

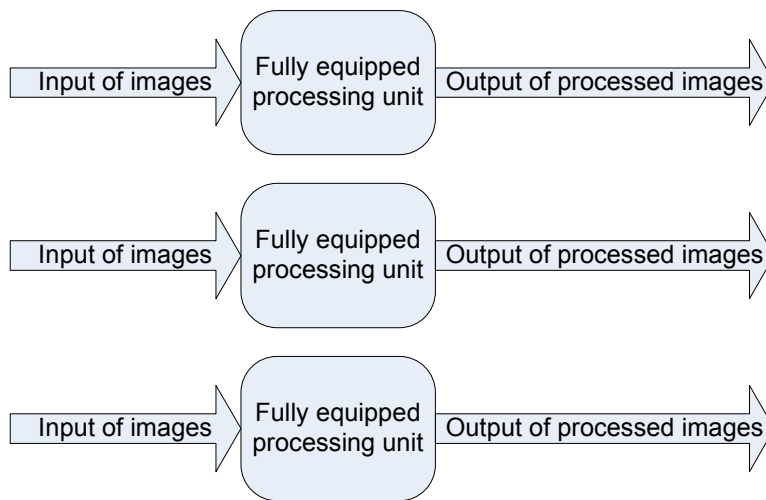


Figure 2.15.: Image Processing with Copied, Fully-equipped Processing Units

This approach is similar to the integration problem, where the entire integration is divided into smaller pieces. These pieces are processed independent from each other. Regarding the images, the problem of applying the filters is not divided into pieces but the input data, the images. One third of the pictures is processed by the first unit, one third by the next unit and one third by the third unit.

For researching different types of parallelisation, another approach is used. If certain parts of an algorithms still have to be executed in sequence, pipelining can be used to parallelise the algorithm.

Pipelining is another basic concept in parallel computation. It is mainly used in advanced microprocessors. The processor starts the execution of the next instruction before the first instruction has been completed. Several instructions are in the pipeline

simultaneously, each at a different stage. The pipeline is divided into segments. If one segment finishes its work, it passes the result to the next segment. At the end of the pipeline the final result is emerged. [HP02]

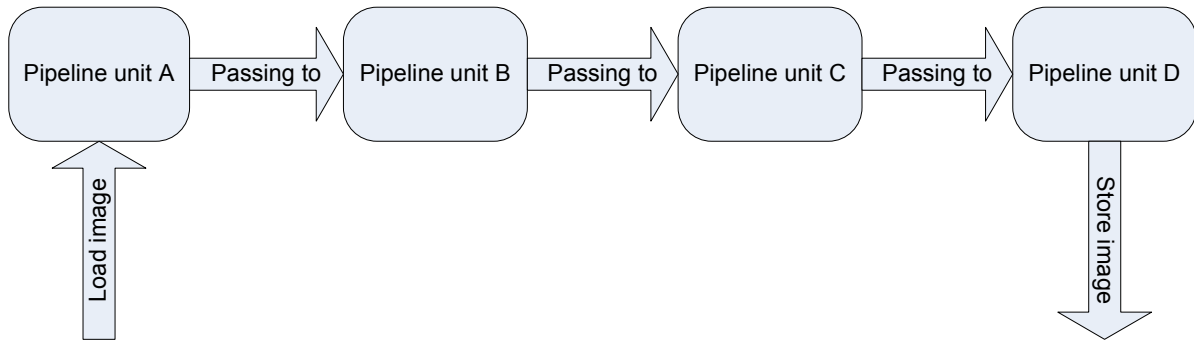


Figure 2.16.: Image Processing Pipeline

Connecting the pipelining with the image processing problem, several units build a chain or pipeline and each picture, which has to be processed, is sent from one unit to the next unit. Figure 2.16 shows such a pipeline.

The first unit reads an image into memory and process it with its appropriate function. After that, it calls the next unit and sends it the image data. In the shown example, the pipeline is filled after four steps and in an ideal case each unit processes a different image at the same time (see figure 2.17 on page 37).

But like a processor pipeline, this pipeline can also have stalls. It is obvious that the slowest image processing function determines the speed of the whole pipeline. In general, each stage of a pipeline should work with the same clock to avoid structural hazards caused by the function units of the pipeline.

This approach to the image processing problem offers possibilities for further adjustments, especially in the directions of fault tolerance and expansions to the usage of multiple pipelines.

Multiple pipelines offer a possibility to parallelise the image processing even more. It combines the pipeline approach together with possibility of having several instances of each unit. Figure 2.18 on page 38 illustrates such a scheme.

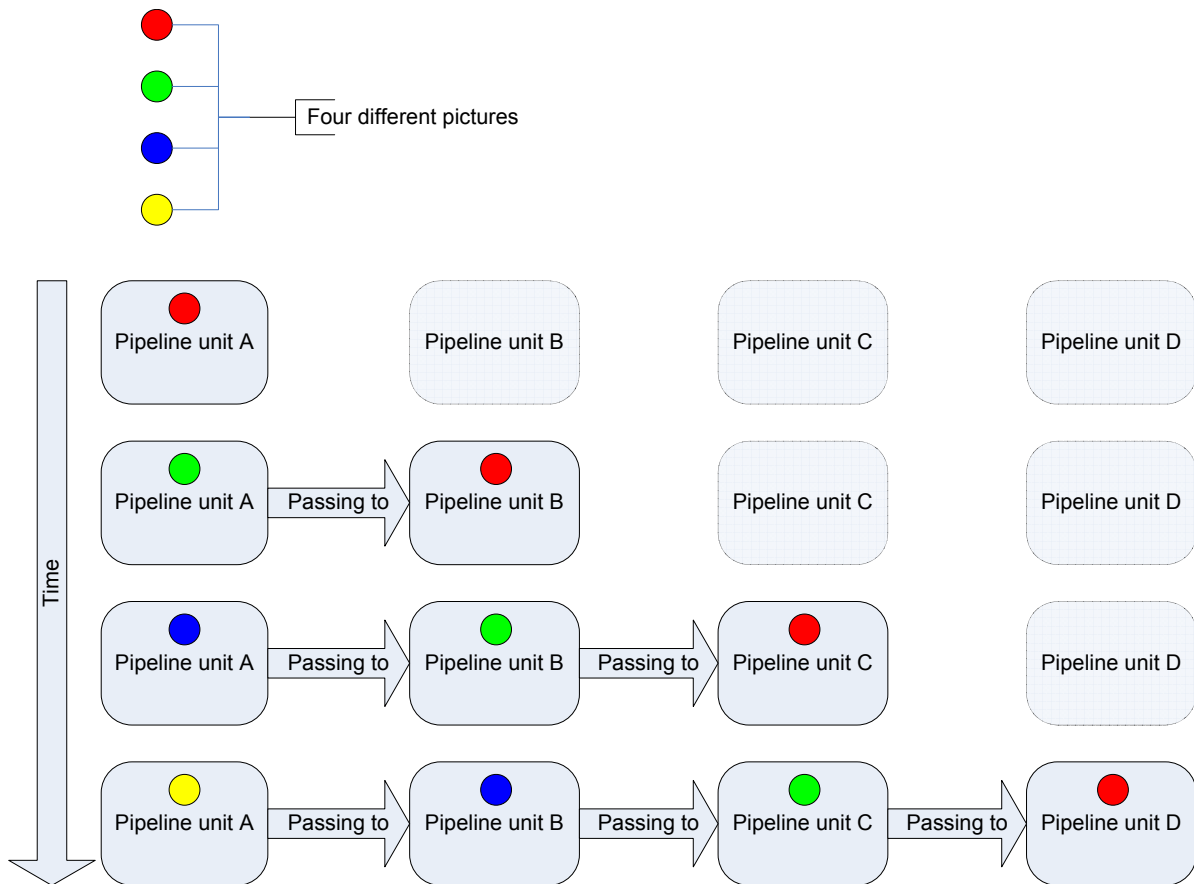


Figure 2.17.: Filling the Image Processing Pipeline with Data

2.6. System Design of a Prototype Parallel Plug-in Suite

The parallel plug-in pattern was created by designing and implementing a prototype parallel plug-in suite. With the help of this suite, the parallel plug-in concept was designed, tested and refined in practice and with the background of practical use cases.

This allows a simultaneous verification of conceptional decisions, as well as easy adoption possibilities by other users, who might reuse parts of a parallel plug-in example or even a whole prototype plug-in for their own purposes and research.

The parallel plug-in suite also combines the advantages of the previously introduced

2. Preliminary System Design

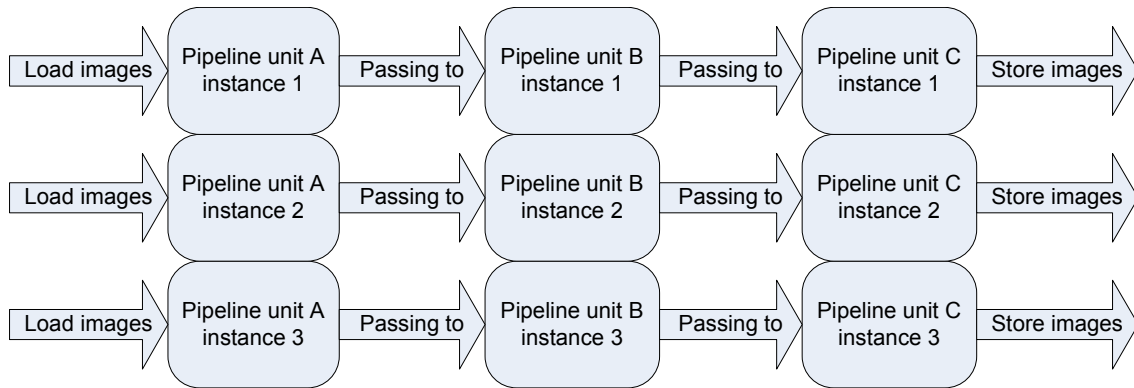


Figure 2.18.: Multiple Image Processing Pipeline

communication basics, the plug-in concept and the discussed parallel plug-in types. A desirable goal was the implementation of a pattern for each of these three explained parallel plug-in types. Possible realisations of an replicated and distributed parallel plug-in are discussed, as well as the layout of a service plug-in.

Thus, the suite is set up with three different plug-in prototypes, each highlighting its particular features and benefits. However, the experimental character must not be forgotten, as the research in parallel plug-ins is still at the beginning. Therefore, several prototypes were generated and expanded with additional features.

The parallel plug-in technology is set up on the Harness runtime environment. The cooperation with the Harness kernel is of vital importance as the Harness functions allow the plug-ins to be executed, to execute other programs, access and communicate with each other, or even to access available resources in general.

Thus, the basic set-up of a plug-in, the plug-in frame is almost identical. The introduction of communication concepts and architectures and the implementation of specialised functions lead to the categorisation of a plug-in into a certain plug-in type.

The main question for the user is the purpose of the plug-in, or what he or she wants to achieve with it. Therefrom, certain design criteria were derived. The following designs and decisions were influenced by such questions.

Furthermore, already existing parallel programming models were integrated. The single program multiple data (SPMD) and multiple program multiple data (MPMD)

models can be highlighted. Today, these models are fundamental programming concepts for implementing parallel algorithms and programs. [Buy99]

SPMD is the dominant style of parallel programming. All processors execute the same program code, but each operates on different data. In MPMD, multiple processes use a single thread so that multiple codes act on multiple data structures. Mostly, these existing programming paradigms are realised with message passing systems like PVM and MPI. Therefore, appropriate communication features were used for this project as well.

2.6.1. General Communication Aspects Regarding Parallel Plug-ins

Communication plays a decisive role. As the parallel plug-ins are distributed applications, communication is needed in almost all phases of the application, starting with the coordinated loading of all plug-in units. Then data and messages have to be exchanged. Especially messages play an important role considering fault tolerance.

Harness allows a unit of a parallel plug-in to be executed, but communication allows the establishment of parallel applications. The necessary communication was performed by the RMIX framework, which was currently integrated into Harness.

The RMIX framework is implemented as an additional plug-in providing synchronous and asynchronous communication. Both kinds of interaction offer different possibilities of application programming. They were used in an appropriate way during the development of the parallel plug-in prototypes.

RMIX was chosen, because it is part of the Harness and parallel plug-in research project, but it also offers all the functionalities, which are needed for a successful communication system within a parallel plug-in. Furthermore, it also informs the user about occurred errors or interrupted connections. This is a very important fact regarding the implementation of fault-tolerant mechanisms.

2.6.2. General Fault Tolerance Mechanisms Regarding Parallel Plug-ins

Usually, fault tolerance consists of three steps. First of all, failures have to be detected. In the Master thesis project, a failure or error in the parallel plug-in application can be equated with the breakdown of one or more of the participating plug-in units.

Failures can be detected during the start of a parallel plug-in or while sending messages between the single units. This shows the dependence on a good communication system, which can forward errors.

Other possibilities for failure detection can be excluded, because the units run on independent Harness kernels which are mostly situated on independent compute nodes. Since the Harness kernel only provides basic functions, the plug-ins may provide fault tolerant services for themselves or for other plug-ins, too. [EG05a]

After detection, notification must take place. Here, all other plug-ins or the leading instance have to be informed, which depends on the used architecture. In a client/server environment, for instance, the client must be informed that the server cannot execute requests anymore. In a peer-to-peer environment at least all peers, which may contact the failed unit, have to be informed.

The notification process is important, as all the remaining units can start a predefined reaction after a plug-in unit failed. Reacting on a failure is the third stage, whereas the reactions can be of different kinds.

If the plug-in units are mostly independent, the work, which was performed by the failed plug-in element, can be redistributed to the remaining plug-ins. This postulates the knowledge of the work or more exactly the calculation interval assigned to each plug-in.

Another possible reaction is the coordinated shutdown of the remaining plug-ins. That can be necessary, if the failed plug-in was of vital importance and a reload of this plug-in is not possible or a redundant system is not available.

2.6.3. Parallel Plug-in for Monte Carlo Integration

The first practical use case is the Monte Carlo Integration. The general goal is a speed improvement for the integration process. The figures in table 2.1 on page 31 show that the computation of high dimensional integrals needs a lot of time.

Besides the approach of using methods like Monte Carlo, the calculation of integrals in parallel reduces time costs. A conclusion might be that the availability of more compute nodes would reduce the time for the calculation more and more as it is a scalable problem.

But the costs of communication have to be considered as well, because data has to be sent between the plug-in units. The sending of information and data via a network also costs time.

2.6.3.1. Design Approach

Neglecting the fact of communication costs, the overall design approach assumes that the availability of more integration units improves the speed of the overall integration process. As many units as possible offer an integration service.

At this place, the client/server approach is a good architecture for solving the problem. Unlike a general client/server application where mostly a small number of servers offer their services to many clients, a lots of plug-in units offer their integration service to one client.

Figure 2.19 on page 42 illustrates the design approach. These single plug-in units together build up a replicated parallel plug-in, which follows the SPMD approach. The SPMD concept is fulfilled as a master program, in the figure represented by the client, distributes the data on several available working nodes, the servers. The intelligence of the whole program is concentrated in this master unit.

It is responsible for the distribution of the data and the collection of the results. Different implementations may allow the master to calculate a certain piece of data as well, which was not planned in this case.

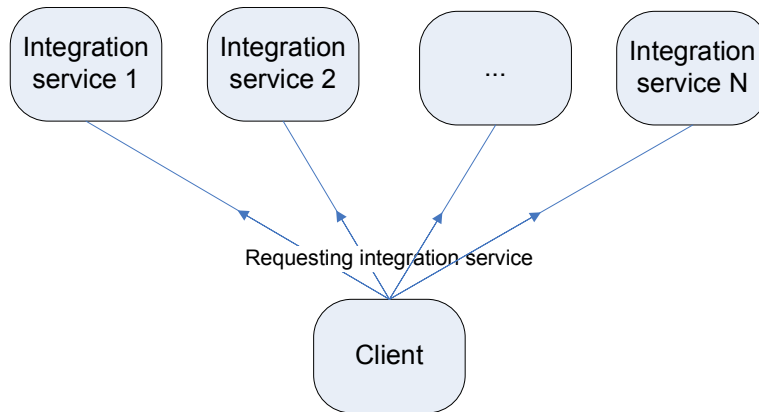


Figure 2.19.: Concept of the Parallel Plug-in for Integration

The aim is a uniform distribution of data on all working machines. Now, all compute nodes perform the same kind of program, the parallel plug-in unit for integration. That is why, it is possible to speak of a replication of the plug-in units. The network of plug-in units offers the construction of a replicated parallel plug-in, performing the parallelised integration service.

From figure 2.19, the topology of the replicated parallel plug-in can also be derived. A star is constructed and the client, the controlling unit, is the center. As the service is only requested by the client, no communication is needed between the single parallel plug-in units. Therefore, the star offers the appropriate topology form.

For the parallel plug-in for Monte Carlo Integration, the client is also in charge to perform the failure detection. While the integration is in progress, it is not possible to detect the failure of one of the processing nodes. The detection of a failure is as recently possible as the collection of the single intermediate results is started. Failed plug-in units cannot be contacted for results.

At this point, the master program has to be equipped with appropriate functionalities for the failure handling. One option is the abortion of the complete calculation and the controlled shutdown. A second, more intelligent option is the recalculation of the missing integral section.

Therefore, the client has to redistribute the missing sections on still available nodes. But this is only possible, if it is reproducible which piece of the integral was processed

by a particular node.

The easiest possibility is that the client stores the names of the nodes together with the sections of the integral in a list. A disadvantage of this option is the waste of memory, especially if a great number of nodes is used.

A second way, which was used in the project, is the assigning of a certain rank to each node. For contacting the plug-in units, the master program must have their names or contact information in storage, for instance in a list. Now, the position in the list can be equated with the rank.

It is assumed that the integral is distributed equally over all nodes, and the integral parts are sent one after another to the nodes in the correct order as the nodes are stored in the list. So, the part of a certain node can be recalculated by its position in the list, and no additional memory is needed for the installation of fault tolerant mechanisms.

2.6.3.2. Data and Functions Overview

Based on the design, different data have to be stored and processed by certain functions. Regarding both, data and functions, there are differences between the parallel plug-in units performing the calculation and the master program coordinating them.

First the data and function needed by the integration units are discussed. The following lists present an overview of the data and required functionalities. The data includes only information for the calculation process, which must be provided by the requesting side.

- Coefficients, representing the integration function
- Integration boundaries, defining the computation limits of the integral
- Random numbers, defining the amount of supporting points used to compute the integral

The functionalities must take care of accepting requests and the integral calculation.

- Initialisation of the plug-in unit
- Communication capabilities for accepting incoming request and unpacking data
- Integration function performing the calculation
- Random Number Generator, generating random numbers for the computation

For distributing the data to the integration units, the client has to be equipped with additional information besides these which are presented in the data list. First of all, it needs the names of the nodes running an integration unit or other equal information, which allows it to contact the unit and request a service.

The functionalities of the client are different from these of integration units. As the client is responsible for the overall calculation process, it needs functions for the distribution of the integral data, as well as for the collection of the results. Further on, appropriate fault-tolerant functionalities must be built in.

- Reading the integration data from an external source, i.e. a file
- Determining the data for each integration unit and send it to it to each unit
- Request all integration units for their computation results
- Detection of failed nodes
- Redistributing integral sections, which calculation failed

2.6.4. Parallel Plug-in for an Image Processing Pipeline

The second use case of parallel plug-ins deals with image processing. Regarding the application, the goal is the reduction of processing time similar to the integration example. Furthermore, the image processing problem was used to investigate the communication possibilities provided by the Harness RMIX plug-in.

2.6.4.1. Design Approach

This example was chosen to be implemented as a distributed parallel plug-in, since each parallel plug-in part provides another image processing functionality. Therefore, the image processing pipeline uses the fundamentals of the MPMD concept. Each plug-in executes different code caused by the use of different image filters.

Besides this fact, the problem also has characteristics of a peer-to-peer approach. Each plug-in unit is client and server at the same time, by providing image filter to other units and also calling filter functions from others. Unlike the parallel plug-in for integration, there is no master program. All the plug-in units are equal.

As it is a pipeline application, the logical topology is a line, which is illustrated in figure 2.20. A bus is not suitable, as it is not necessary to send an image from one unit to all the other units simultaneously.

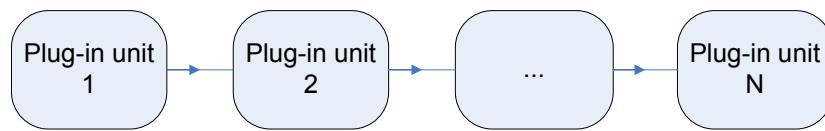


Figure 2.20.: Concept of a Parallel Plug-in for Image Processing

Regarding figure 2.20, one recognises the fact that the first units needs functionalities for reading in images, as well as the last unit appropriate storing functions. Activated by a certain impulse, the first unit starts reading images, performs the first filter on the data, and calls the filter function provided by the second unit. So, the images are handed over until they are stored by the last unit.

The first unit, loading the images, noticed the fact that all the image were loaded and could terminate, but how does this will be detected by the other parallel plug-in units? Otherwise they would wait and listen for incoming calls forever.

To prevent the plug-in units from running forever, the number of images is sent to each participating unit before the direct processing begins. Furthermore, an additional communication capability is built in. Figure 2.21 on page 46 shows that there is now a new communication way from the last plug-in unit to the first one passing all the other units.

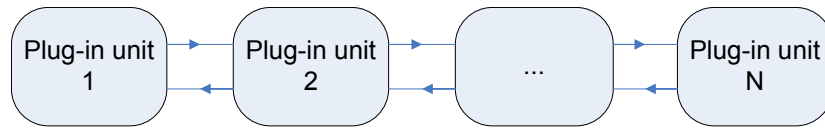


Figure 2.21.: Image Processing Pipeline with Acknowledgment Functionality

If the last plug-in stores an image, a message is sent to the previous plug-in, containing the information that one image was successful processed and stored. This message is forwarded unit by unit until it reaches the first one. Now, each plug-in has the information about the number of images overall and the number of processed images. If all images were processed, the plug-ins can terminate.

The installation of the back way communication is also the first step of including fault-tolerant mechanisms. Like in the integration example, the fault-tolerant mechanisms consists of the three steps detection, notification and reaction.

Detection is only possible by contacting another plug-ins for their services. Regarding figure 2.20 on page 45 the successor plug-in units would not detect the breakdown of the first plug-in, but with the back channel in figure 2.21 this use case is solved.

Now, failures can be detected and the notification must be considered, as well as suitable reactions. The participants, which are directly influenced are the predecessor and the successor plug-in unit.

Based on the fact, that all images have to be processed by the same filters in the correct order, there exists only one option. The plug-in has to be reloaded.

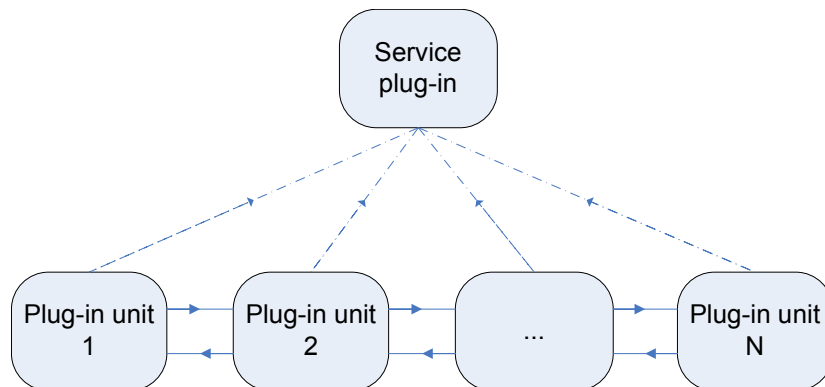


Figure 2.22.: Image Processing Pipeline with Service Plug-in

In figure 2.22 on the previous page, a service plug-in is introduced which is responsible for the reloading of plug-ins. If a failure is detected, the service plug-in is called, and it tries to reload the failed plug-in on another node. A form of by-pass is built.

The next step after reloading the plug-in is the update of predecessor and successor plug-in units, because they need the new contact information. If the plug-in cannot be reloaded, the whole pipeline must be terminated, as the images cannot be processed with all filters anymore.

Another problem is the possible loss of images, which were just processed by the failed plug-in. Therefore, each plug-in has a list with images processed by it. An image is added to the list, when it is processed and deleted from it, if the notification from the successor plug-in arrives, containing the information that the image was stored successfully.

When a plug-in is reloaded, it is now possible to identify the lost images on the basis of the image lists of the predecessor and successor plug-in units in the pipeline. These images must be reprocessed. In the case of the image processing pipeline, fault tolerance is traded against memory storage.

2.6.4.2. Data and Functions Overview

First of all, the data of the image is the most important one. There are two different possibilities to access this data. One of these possibilities is based on the availability of a network file system (NFS). Here, the files are not transported from one node to another, but they can be accessed remotely as if they were stored locally on the hard disk.

A disadvantage is the scalability of the access to an image. If several plug-ins try to access images, they just want to process, the server, storing the images, have to support a lots of requests. Therefore, the complete image data is transferred from plug-in unit to plug-in unit until it is finally stored by the last one.

Other information needed for a successful processing of images are:

- Image data
- Size of the image
- Directory with the images, which have to be processed
- Directory name, in which the processed images have to be stored
- Contact information of the predecessor and successor plug-in
- Number of images at all
- Contact information for the service plug-in
- Name of the crashed plug-in (only service plug-in)

The functionalities have to take care of accepting requests and the image processing.

- Initialisation of the plug-in unit
- Communication capabilities for accepting incoming request and sending a request to the next plug-in unit
- Image filter functionalities
- Communication capabilities for requesting the service plug-in because of a broken pipe
- Loading images (first plug-in unit in pipeline)
- Storing images (last plug-in unit in pipeline)
- Capabilities of reloading a plug-in (only service plug-in)

2.6.5. Parallel Plug-in Manager

Parallel plug-ins are not automatically running on a set of compute nodes. They have to be loaded and initialised. Derived from figure 2.11 on page 29, there exist different options. As it is not convenient for a user to do this for every node, an automated process is preferred.

On the one side, there is a pool of available machines and on the other side, there is the user who wants to run his parallel plug-in applications. Now, a service plug-in can handle the loading. The user must only start this service plug-in and it automatically loads the desired parallel plug-in.

First of all, the (un)loader plug-in has the task to load a replicated or distributed parallel plug-in on a specified number of Harness kernels. In addition to the loading, it also manages the unloading of each plug-in unit, if an application was finished. In the course of this thesis, the parallel plug-in loader is named PPM (**P**arallel **P**lug-in **M**anager).

2.6.5.1. Design Approach

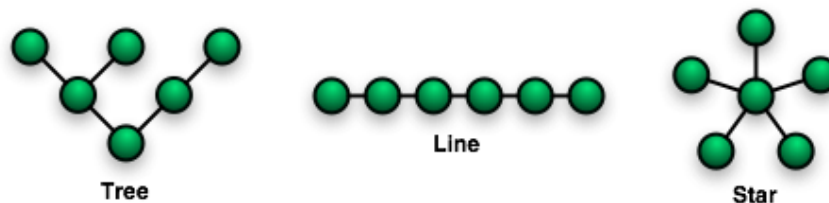


Figure 2.23.: Possible Ways of Loading Parallel Plug-ins

The PPM can be implemented in various ways to load parallel plug-ins, figure 2.23 shows different possibilities. First, a tree structure could be used. The loading is performed in a cascade way. The PPM can initiate the loading of the first plug-in units. After that, these units will load the next level of parallel plug-in components. This cascade is repeated until all parallel plug-in units are loaded.

This option raises different problems. If one of the first plug-ins fails, a whole sub-

tree may not be loaded. Difficulties regarding the integration of fault tolerance occur. An advantage of this option is the time factor, as the loading is performed not only by one instance but by all instances in a same level of the tree structure.

A second possibility is a chain. The service plug-in loads and initialises the first plug-in. Then, this can initialise the next one and so on. In general, this option is one of the most unpractical solutions. It is time consuming and all information including the remaining plug-in units and the initialisation parameters have to be transferred to the next, just loaded plug-in.

The third option, which was implemented, has a star structure. The PPM is the central point and loads the whole parallel plug-in. If broadcasting is available, the PPM can send loading instructions to all nodes simultaneously. These loading instructions include the starting request for a Harness kernel and the automated loading of the plug-in unit.

Furthermore, in a star configuration the PPM is the central point of reference for occurring problems. It detects the non-loading of a unit and may inform the user about the status, for instance via a log file, or directly via printouts on the screen, as it can be assumed that the user is operating the node where the PPM was loaded.

Besides the loading, the PPM can also be responsible for the unloading. Special notification possibilities can be integrated notifying the manager that the parallel plug-in application has been finished, and the parallel plug-in components can be unloaded maybe connected with the shutdown of the entire Harness kernel. At this place, the same configurations can be used as for the loading (see figure 2.23 page 49).

Fault-tolerant mechanisms can be integrated in different variations as well. They have to be customised at the specific situation and the needs of the specific application. For example, the two introduced applications, the integration and the image processing pipeline, demand different levels of fault tolerance.

The integration is a scalable problem, as the integral is distributed on the available nodes. If at least one plug-in unit can be loaded, the computation can be performed. For achieving the computation result in an appropriate time, a certain minimum of needed compute nodes could be defined. If this minimum is not reached, the appli-

cation would not be started.

In this context, the term of partial success can be introduced. Partial success deals especially with problem of non-loading all desired parallel plug-in components. In case of a partial success, the integration application can still be executed. This is not possible for the image processing pipeline.

In the pipeline example, a partial success has to trigger predefined actions by the PPM. The application requires the loading of all parallel plug-in components, as each of them makes its contribution to the image processing. Possible reactions are the abort by the PPM or much better the PPM tries to reload the failed plug-in unit(s).

Therefore, a pool of redundant compute nodes is necessary. If there are still free nodes available, the PPM attempts the reloading on one of these machines, otherwise the entire parallel plug-in application must be aborted and already loaded plug-ins must be unloaded.

Besides the (un)loading, it is also conceivable that the PPM can perform additional tasks, i.e. providing further services. The manager can be integrated in the fault-tolerant mechanisms of parallel plug-ins, for instance, it can be responsible for reloading plug-in components during the execution as it has the information of still available nodes. It may also be involved in the coordination of plug-ins including internal work distribution.

2.6.5.2. Data and Functions Overview

For performing its tasks, the PPM needs different input information, for instance

- Name(s) of the available nodes
- Plug-in(s), which must be loaded
- Node of the PPM
- Switch for choosing between different plug-ins

The names of the nodes have a form of a list and in the case of a replicated parallel plug-in the mentioned unit is tried to be loaded on all of the machines in the node list. If a distributed parallel plug-in has to be loaded, the listed plug-in units are loaded, each on another node, and nodes remaining free are added to a pool of still available machines.

If the PPM is integrated in a parallel plug-in and plug-in components must contact the PPM, the node on which the PPM is running must be known. Furthermore, the manager can be extended for loading more than one parallel plug-in. Here, a switch is needed to decide which parallel plug-in is loaded.

Regarding the functionalities, the main ones are listed below

- Initialisation
- Loading replicated parallel plug-ins
- Loading distributed parallel plug-ins
- Unloading parallel plug-ins
- Fault tolerant mechanisms

Especially the fault-tolerant mechanism must be adjusted to the desired purposes of the plug-ins including facts like partial success.

3. Implementation Strategy

3.1. Implementation and Integration Strategies

The implementation of the prototype parallel plug-in suite had to fulfil different requirements. Scientific example applications had to be integrated and the research in parallel plug-ins had to be advanced as it is a new research field. Examples were well chosen, to include as many Harness features as possible. As the parallel plug-ins were loaded into the Harness runtime environment, their implementation was similar to programming paradigms used by Harness.

Furthermore, a target was to show scientists and programmers different options of implementing parallel plug-ins, but also possible problem areas were shown. Nevertheless, an aim was the creation of reusable prototypes and guidelines for parallel plug-in programming.

First, simple plug-in prototypes were implemented, which were loaded into Harness kernels. After proving these basic features, the prototypes were augmented with functions for communication. Here, the client and server stubs had to be implemented. The stubs were responsible for the packing and unpacking of parameters for the remote functions.

The communication capabilities led to parallel plug-in prototypes for the Monte Carlo and image processing applications. Furthermore, failure handling mechanisms were added in further implementations phases, as well as the research in the use of the Harness thread environment for image processing in multiple pipelines.

3.1.1. Programming Issues

The basics and interfaces, which connect the parallel plug-ins and the Harness runtime environment, are based on the Harness project sources developed by the Oak Ridge National Laboratory. This Harness prototype was implemented in C. Therefore, the parallel plug-in prototype suite was realised in C as well.

The advantage of using C as programming language is the easy connection to the existing Harness RTE interfaces and the preferences of this language including the better performance by using operating system calls directly and its support of the data types demanded in the project.

C does not need any additional virtual machine, such as Java, and allows the direct access to system calls. This provides more performance, but it also puts the responsibility for good programming on the user. For instance, C allows it to access memory directly without additional type controls and the programmer has to take care that no forbidden memory spaces are accessed neither reading nor writing.

The programmer is also responsible for freeing any resources which are not used anymore. Besides this higher responsibility while the programming, C may also support the realisation of first prototypes as even unconventional programming methods could be used.

C provides flexibility, efficiency and heterogeneity. It maintains unique data types for stored and transferred data and the flexibility is provided by a modularised design approach and the partition in functions. New program features can be added by integrating additional functions. The design can also be enhanced and improved with new modules.

3.1.2. System Environment

This parallel plug-in project had software, as well as hardware requirements regarding the system environment. The use of parallel plug-ins premises the availability of a Harness runtime environment and an appropriate Harness RMIX library. The Harness RTE includes the Harness daemon and the Harness library, which contains the

kernel and the interfaces to its functions.

As Harness is always improving, the newest version can be found at [Eng]. This package also includes the current version of the RMIX library, which provides the communication features.

Furthermore, the system has to provide an ANSI-C compiler and the ANSI-C libraries to compile the sources. Additionally, two libraries must be installed. The first library is the libConfuse [Hed04], which is responsible for reading in configuration files and the second library is ImageMagick[LLC05], which provides filters and functionalities for image processing.

The hardware consists of at least five dual Pentium IBM-compatible workstations equipped with 512MB RAM and 500MHz. The necessary hard disk space depends, i.e. on the number of images, which will be processed. All the more images have to be processed all the more memory is used. The workstations are connected via a TCP/IP network, whereas the links between the nodes are established by using a hub.

Furthermore, the prototype suite was developed under the Linux operating system Debian Sarge 3.1 by using the C compiler gcc 3.5.3 and the autotools. During the test of the components and the whole prototype suite, the Harness and parallel plug-in processes were started and executed in a command line user interface.

3.1.3. Parallel Plug-in Manager

The basis for the loading of parallel plug-ins is the knowledge about their names and the available nodes. Because of the fact that it is not possible to append parameters to a plug-in, which is loaded into the Harness RTE, it is necessary to use another way to provide the plug-in with the required information.

The plug-in loading function, which binds the plug-ins to the RTE, is not yet capable to pass parameters to the new loaded plug-ins. Therefore, this problem was solved by using configuration files. After binding the PPM plug-in to the RTE, it searches in a certain path for a configuration file.

3. Implementation Strategy

This file contains all the needed information. An example for this file can be found in appendix A.1.2 at page 104. The file is read in with the help of the libConfuse library. This C library is written for parsing configuration files. It supports sections, primitive data types and lists of primitive data types.

The use of a library also offers a simple extension of the configuration file. The information read out of a file is stored in lists. An advantage of a list is the navigation in it, as a list can be passed through fast, and the deletion of elements does not cause any gaps as if deleting elements in an array.

After the analysis of the configuration file, the RMIX library must be loaded, as it provides access to the communication features. Afterwards, the loading of other plug-ins can be started. It is assumed that there are no loaded Harness runtime environments on any of the available machines. Therefore, a connection has to be established to each node and a Harness kernel together with the desired plug-in unit is started.

The Harness kernel function for executing an external process is used, whereas the link is established via a secure shell (ssh) application. The ssh link is opened, and the command for starting a Harness daemon together with the parallel plug-in unit is executed on the remote side. That is why, it is also not possible to transmit any parameter for the plug-ins at this stage.

After finishing the parallel plug-in application, or if a failure occurs during the loading, the Harness RTE including all loaded plug-ins must be terminated. Hence, two possibilities were implemented, which use depends on the stage in which the parallel plug-in unit is.

If the Harness RTE has a loaded RMIX library on the remote side, the internal function for the shutdown of the kernel can remotely be called. If the kernel runs without any communication capabilities, a ssh connection must be established and a signal is sent to the operating system to terminate the Harness RTE including all loaded plug-in components.

3.1.4. Replicated Monte Carlo Integration Plug-in

The implementation of the Monte Carlo integration plug-in was organised in several steps. First of all, the mathematical algorithm was programmed. This happened in the form of a single console application. The random generator included in the standard c library was used as a pseudo random number generator.

Each random generator is suitable for a special set of tasks. But the standard c library is not appropriate for the creation of numbers in statistics and numerics, such as the Monte Carlo methods. The generator does not fit the necessary requirements, but here the choice of the generator is not so important as the main focus is on the programming of parallel plug-ins. [Kom05]

The next step was the integration of the algorithms into a plug-in frame, which can be loaded into a Harness RTE. At the end of this implementation phase, it was possible to load a plug-in locally, and it performed integral computations. Therefore, the integration data had to be read in. For this data, a file was used similar to the file for the PPM configuration. An example for such a file can be found in appendix A.1.2 at page 104.

The third implementation stage dealt with the integration of communication capabilities and the construction of a parallel application. Derived from figure 2.19 on page 42, the Monte Carlo integration parallel plug-in consists of at least one server plug-in providing the integration features and a service plug-in asking for this service.

In the prototype suite, the service plug-in and the PPM plug-in are one and the same. The PPM was extended with features for reading the integral input data and asking for integration services. Figure 3.1 on the next page 58 shows the general life cycle of the Monte Carlo integration application.

After loading the PPM, it reads the configuration file. Then, the parallel plug-in components are loaded via the internal Harness call for executing an external process. On the side of the parallel plug-in component, the Harness plug-in loader also calls the plug-in component's initialisation function. This function is responsible for the export of the local RMIX plug-in and the export of the integration service on the remote side so that requests can be accepted.

3. Implementation Strategy

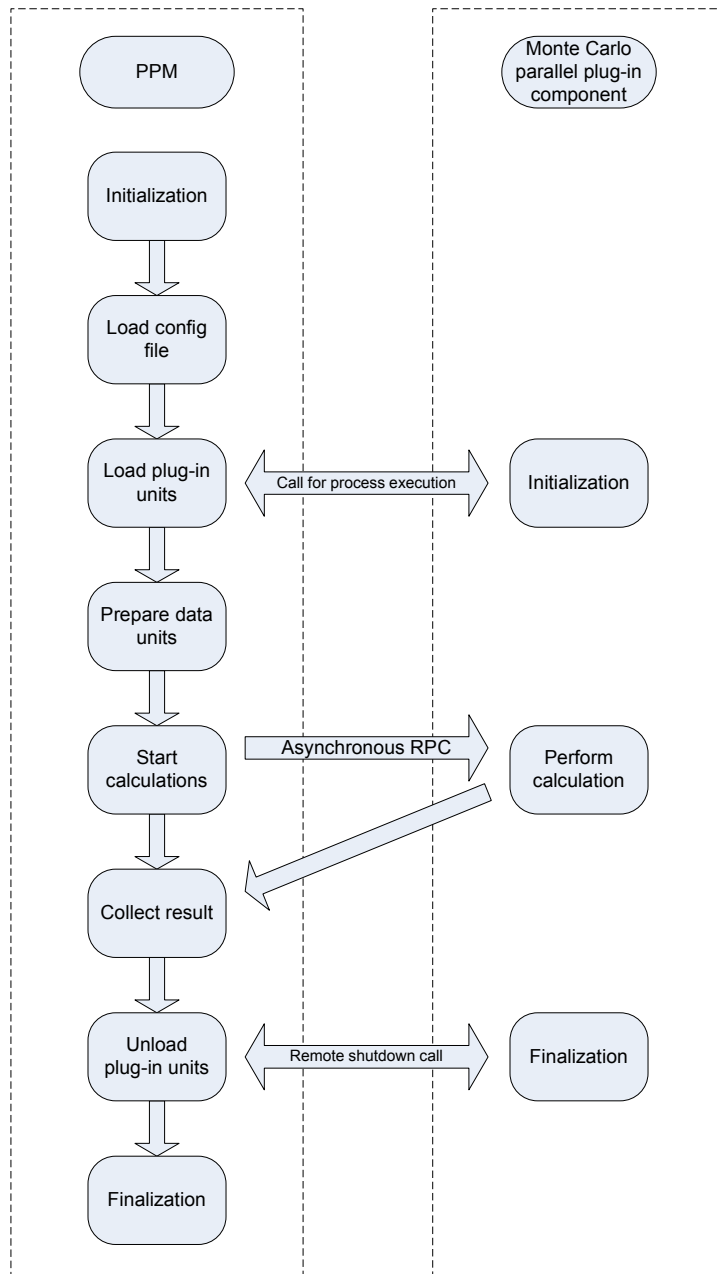


Figure 3.1.: Life-cycle of the Monte Carlo Integration Application

3. Implementation Strategy

Then, the PPM prepares for each parallel plug-in unit the input consisting of the particular interval, the coefficients from the equation and the amount of supporting points to be used. This information is sent to the plug-ins, whereas asynchronous RPC calls must be used. Otherwise, the PPM would be blocked until it gets the result from the called plug-in function.

As all plug-ins calculate in parallel, the integration functions must be called one after another and the results are fetched later. After collecting the results, the PPM unloads the parallel plug-in components. Here, the remote Harness function for the shutdown of a kernel is used. After the termination of all parallel plug-in units, the PPM may terminate itself.

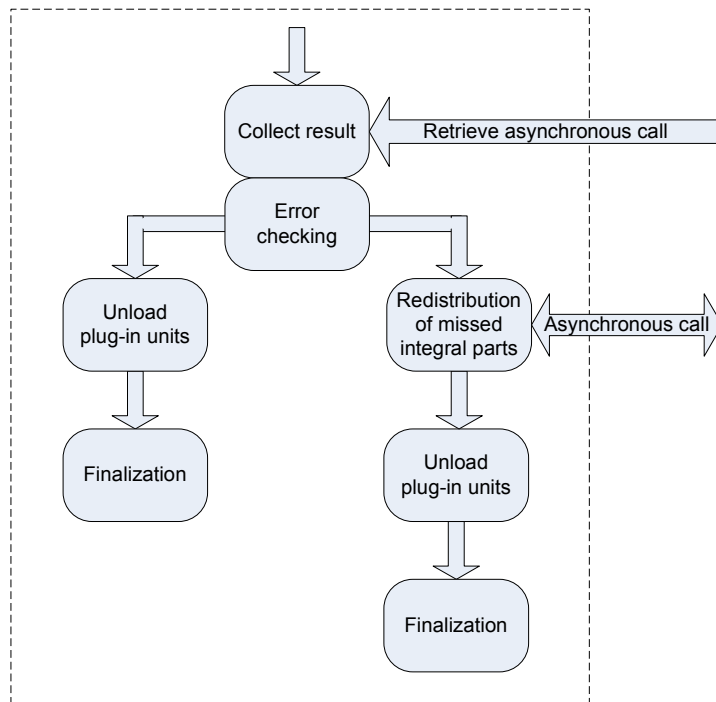


Figure 3.2.: Monte Carlo Integration with Additional Failure Handling

In the last implementation step, additional fault tolerant features were added. While fetching the results, the PPM detects plug-in units, which are not reachable anymore. So, some results may miss. In accordance with the available list of Harness kernels, the missed calculation results are redistributed on the remaining plug-ins.

Therefore, the PPM recalculates the part of the integral, which was performed by the

failed unit, and as long as there are operative nodes left, the missed result can be computed. According to figure 3.2 page 59, the function of the PPM for collecting the results is amended with additional functionality. Concerning the possibility that more than one parallel plug-in unit could fail, the redistribution is performed with asynchronous RPC requests as well.

3.1.5. Distributed Image Processing Pipeline Plug-in

Like the parallel plug-in for integration, the prototype for the image processing pipeline was also implemented in certain steps. First of all, a simple command line application was developed. This program was used for integrating and testing different image filters, as well as loading and storing functionalities of the used image processing library.

For processing the images the ImageMagick library was used. The library offers a C application programming interface (API) called MagickWand, which includes functions for simple image processing. It can create, edit and compose images. Furthermore, images can be cropped, colours can be changed, and various effects can be applied.

The next implementation phase dealt with the integration of the filters into a plug-in frame body. At the end of this phase, a plug-in was loaded into the Harness RTE capable of loading images, applying various filters and storing processed images. This plug-in prototype established a basis for the distributed image processing pipeline.

The goal of the third step was the realisation of a simple pipeline application, which patterned the model from figure 2.20 at page 45. To simplify the implementation process of the distributed parallel plug-in, there was no implementation of several plug-in units with different source code for each used image filter.

Only one plug-in was created, which offered all the features but only one function was used per loaded plug-in. The functionality provided by a certain parallel plug-in components was preset during the initialisation process. For the invocation of these features, client and server stubs were implemented as well.

3. Implementation Strategy

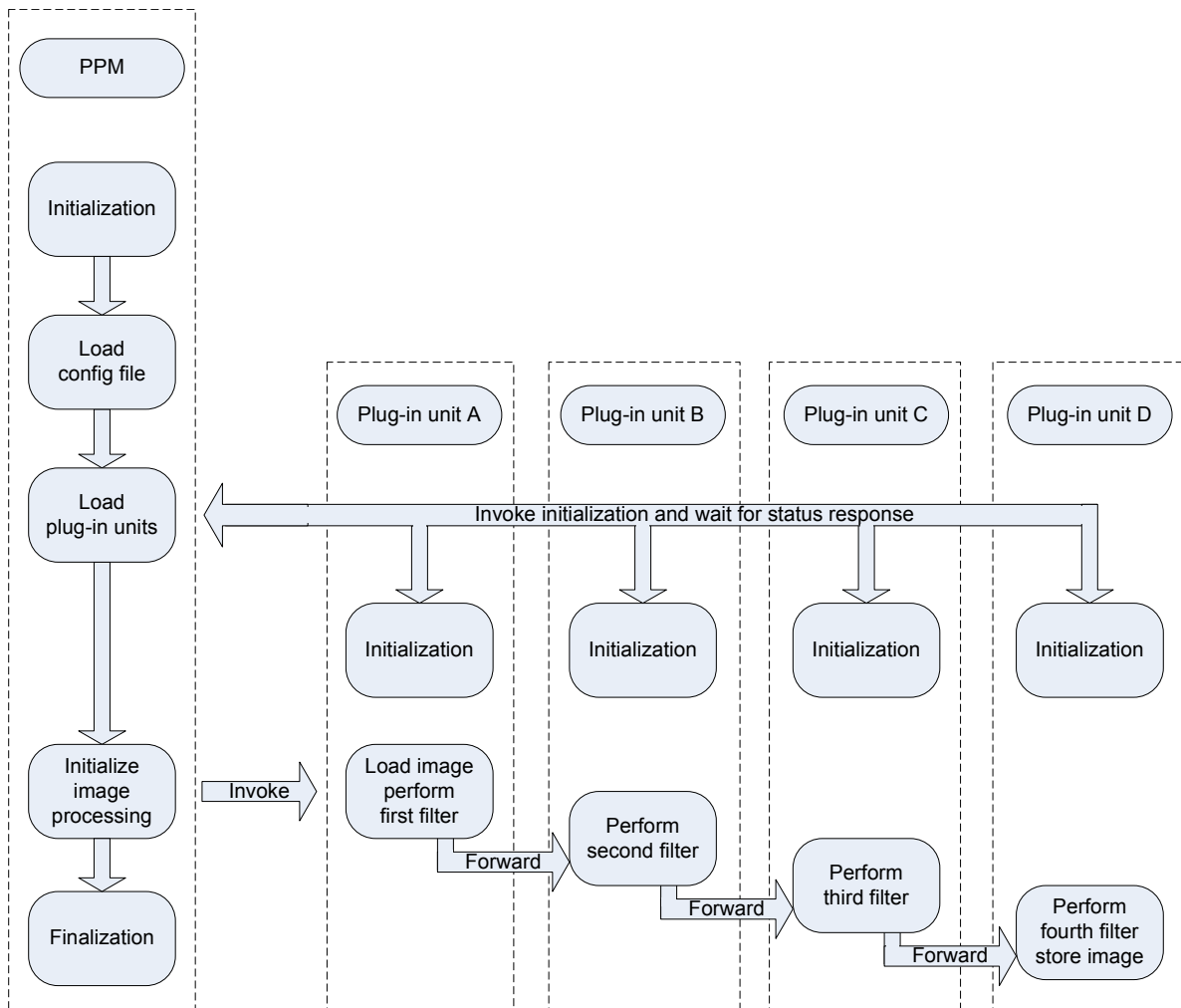


Figure 3.3.: Life-cycle of the Image Processing Pipeline Application

Figure 3.3 at page 61 illustrates the life-cycle of a simple image processing pipeline implementation. In this case, the PPM has only to load and initialise the distributed parallel plug-in units. It also reads in the configuration file and loads the appropriate number of plug-in units. An example input file can be found in appendix A.1.2 at page 104.

As all distributed plug-in units are necessary to solve the image processing problem, the PPM has to verify the correct start of each plug-in. But this checking was included in the next implementation stage. If there were any errors during the current prototype implementation of the parallel plug-in, the execution of the distributed ap-

3. Implementation Strategy

plication was aborted.

After the plug-ins are loaded, the PPM invokes the plug-in with the first filter. This plug-in is also responsible for loading the images. After that, the pipeline is loaded and the plug-ins communicate with each other. No further actions of the PPM are necessary in this integration step, so it may finalise itself. The distributed plug-in runs until the first plug-in component has loaded all images found in the specified folder. Consequently, no further images can be loaded and passed through the pipeline.

Regarding the loading and storing of images another issue had to be considered as well. As the first plug-in loads the images, it is assumed that the image data is stored on that node. While processing the images, the entire image information including name, image size and image data are sent from one plug-in to the next. The use of an intermediate NFS server would cause a lot of traffic, since several plug-ins try to access the pictures for storage.

It is the matter of a scalable problem, as if a lot of plug-in components try to access the memory at the same time, the NFS server could become overcharged. Now, the planned solution of sending all the image information causes increased communication cost between the units in return.

Then, the prototype of the distributed parallel plug-in was augmented with fault tolerance. Therefore, the model presented in figure 2.22 on page 46 was implemented. The first great change affected the loading of the parallel plug-in components. This function of the PPM was extended with an explicit check for the availability of the plug-in unit by calling it with a synchronous RPC call directly after the loading.

The second change dealt with the failure of pipeline components during the image processing. In this case, the planned redirection of the pipeline via the reloaded plug-in unit was implemented with the help of the PPM. The PPM did not finalise itself after the invocation of the pipeline process, but it was augmented with a service function.

While performing the tasks, a failure of a plug-in unit is recognised by the predecessor plug-in. If a plug-in cannot forward the image data to the next one, it informs the PPM by calling its service function. Now, the PPM reloads the failed plug-in. The

3. Implementation Strategy

new address of the reloaded plug-in is returned to the plug-in, which called the PPM service function. After the reloading, the new plug-in gets the address of its successor from the PPM and the pipeline is again closed. Each parallel plug-in component has the appropriate client stub function to call the redirection service of the PPM.

Another feature of the Harness runtime environment, which facilitates the implementation of the image processing pipeline, is the integrated thread pool approach. If a parallel plug-in unit was started on a node and a request was accepted, this request is processed by a new created thread. This is done for each incoming request.

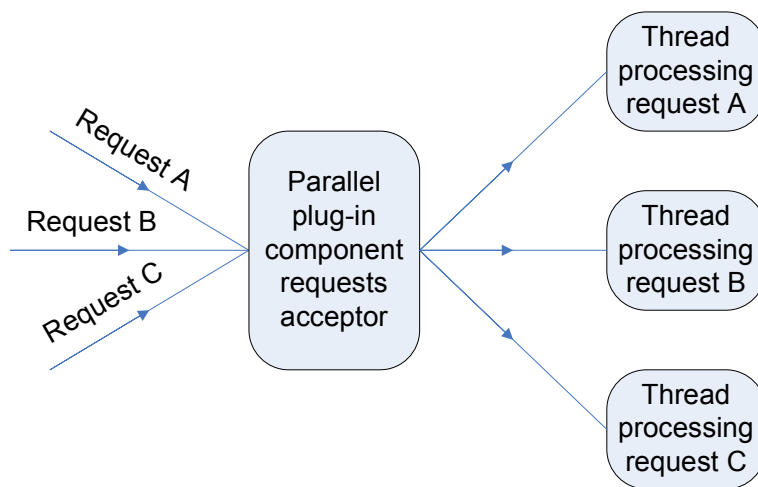


Figure 3.4.: The Harness Thread Pool and Parallel Plug-ins

As illustrated in figure 3.4, it is possible to handle several requests simultaneously. The availability of thread technologies facilitates the realisation of a multiple pipeline, like presented in figure 2.18 on page 38, whereas it is not necessary to load several instances of the same plug-in components because of the automated thread processing.

The thread environment had also to be considered for operating a single pipeline, otherwise it would be immediately a multiple one. To prevent this, a thread mutex were used. The plug-in can accept the requests but only one of the created threads can be executed at the same time. A scheme is presented in figure 3.5 page 64.

The whole section of image processing is now a critical section. Controlled by the mutex, only one thread can perform its image processing algorithms. After the current thread finishes the request and forwards the image, the next waiting thread can

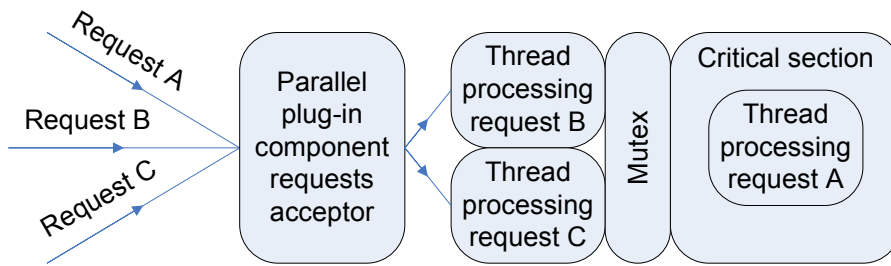


Figure 3.5.: Thread Approach with Critical Section

enter the critical section.

The implementation of multiple pipelines is integrated with the thread pool, whereas it is not a fully multiple pipeline. The first unit of the pipeline only works with one thread. This simplifies issues like load balancing, the equal distribution of the input data as well as the fact that the PPM has to invoke the pipeline only once.

The final implementation scheme of the distributed parallel plug-in for an image processing pipeline is outlined in figure 3.6 page 64. The PPM and the first parallel plug-in component are each performed by one thread. Starting with the second component, multi threaded processing takes place. Finally, the threads of the last parallel plug-in component store the processed images in memory.

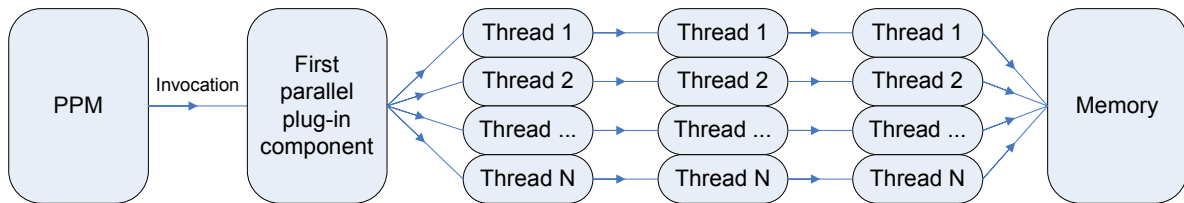


Figure 3.6.: Image Processing Pipeline Implemented with Threads

3.2. Testing Strategies

In software engineering, tests are very important for the verification and validation of software. Testing deals with the quality assurance of new or changed software. The state of software has to be evaluated including the correctness and completeness. The

actual behaviour is analysed by using appropriate test cases, whereas the reaction of the software during the tests is compared with the requirements to the program.

The overall goal of software testing is the search for errors or nonconforming behaviour. Experiences are earned and possible problems in the software can be found. Therefore, a software testing phase during the development of a program is often followed by a review of the system design or even system design changes may be necessary, based on the results of the experiments and tests.

The prototype character of the parallel plug-in applications demanded well planned tests not only concerning the internal correctness of the functionalities, but also the integration into the existing Harness RTE was of vital importance. It was necessary to use different testing techniques to verify the program correctness under various circumstances.

Often, the overall goals of the used testing techniques are different as well. Some techniques search for errors or failures in the algorithms other can be used for testing the cooperative work with already existing software. Regarding the implementation of the parallel plug-in prototype suite, tests had to be performed to prove the functionalities of the major components PPM, Monte Carlo integration and the image processing pipeline application.

An option for testing the correctness was the use of example cases, i.e. the calculation of a certain integral and the processing of a number of images. In general, the design and algorithms were tested during their development on paper. The realisation of the described implementation steps involved the tests of new added features, for instance the loading of a plug-in, the communication between two plug-in components, or fault tolerant mechanisms. Here, different test scenarios were applied. Finally, if all desired features were integrated into the prototype suite, system tests were performed.

3.2.1. Component and Integration Tests

Component tests evaluate single functionalities and modules of the software. As software projects can consists of many modules and a lot of sources, it is often not possi-

3. Implementation Strategy

ble to evaluate the whole system with all included functions during the system test. Therefore, component tests offer the possibility of extracting single functions.

These functions can be tested in detail by sending input data into the module using the defined interfaces. Now, it can be evaluated whether the function or module processes the data in the planned way and the required output data or reactions are produced by the tested module.

Integration tests deal with the connection of the new designed and implemented software into already existing ones. Especially the correctness of the interfaces must be tested. In the present case, the integration of the parallel plug-ins into the Harness RTE as well as the interface connection to RMIX communication facilities were evaluated.

The prototype suite consists of three major components, the parallel plug-in manager, the integration application and the image processing pipeline. The general component tests cover the evaluation of functionalities, which are included in all of the three major components.

For instance, a test deals with the loading of a single plug-in into a Harness RTE, which must be fulfilled by all the three components. Other functionalities are the reading of configuration and input files and the internal storing and processing of these information.

Each major component has specific functions, which also must be tested. Here, functionality tests were prepared, which were often evaluated in small external programs accessing the defined interfaces of the functions. The following tables show tests of functions and expected results.

| Nr. | Feature | Description and expected results |
|-----|--|---|
| 1 | Reading a configuration file | Tested by reading in an example file and verified by printing out of the read information on the screen |
| 2 | Converting configuration file into linked list | Reading in an example file, whereas the converted linked list is printed out on the screen |

Continued on next page

3. Implementation Strategy

Table 3.1 – continued from previous page

| Nr. | Feature | Description and expected results |
|-----|----------------------------------|---|
| 3 | Processing of a linked list | Adding information to a linked list, deleting entries, searching for entries and the deletion of the entire list, whereas the linked list is printed out after each intermediate step so that each step can be reproduced |
| 4 | Loading a plug-in frame | Integrating a plug-in into a Harness RTE, whereas the correct loading is verified by a print instruction in the plug-in frame demonstrating the execution of the initialisation function of the plug-in |
| 5 | Export of a Harness RMIX library | A Harness RMIX plug-in is exported to access its communication features, the correct exportation is verified by executing a function for the generation of remote references |

Table 3.1.: Tests for General Components

| Nr. | Feature | Description and expected results |
|-----|---|---|
| 1 | Loading a Harness kernel remotely | A ssh connection to another node is opened via the process execution function of the kernel and a Harness daemon is started. The verification is funded by the examination of the process table on the remote node indicating the new daemon process. |
| 2 | Loading of remote plug-ins with fault tolerant mechanisms | The configuration list is added with nodes, which cannot be resolved, a appropriate error message of the PPM is expected |
| 3 | Loading of a distributed plug-in with fault tolerant mechanisms | The configuration list is added with nodes, which cannot be resolved, a appropriate error message of the PPM is expected as well as the reloading of the specific plug-in on another available node |

Continued on next page

3. Implementation Strategy

Table 3.2 – continued from previous page

| Nr. | Feature | Description and expected results |
|-----|---|---|
| 4 | Unloading a Harness kernel via an external process execution | A ssh connection to another node is opened via the process execution function of the kernel and a signal is sent to the remote system to terminate the Harness kernel including all plug-ins, for the verification the process table of the remote node is examined |
| 5 | Unloading a Harness kernel via an internal Harness kernel shutdown function | A RMIX connection to another node is opened and the internal kernel shutdown function of the remote Harness RTE is called. |
| 6 | Reading in the input file for the integration application | An example file is read and its content is printed out on screen |
| 7 | Scheduling the integral | A dataset for an integration is divided into a preset number of parts, the integration intervals of each part are printed out |
| 7 | Reading in the input file for the image pipeline application | An example file is read and its content is printed out on screen |
| 8 | Initialising the components of the image processing pipeline | The initialisation is simulated by printing out the information, which would be send to each component of the pipeline for its initialisation |

Table 3.2.: Tests for PPM Components

| Nr. | Feature | Description and expected results |
|-----|---|--|
| 1 | Computation of an integral | A pre-calculated example integral is computed by the integration program using the Monte Carlo algorithm, the result of the computation is compared with the pre-calculated result |
| 2 | Computation of the function value of a function | The function values of a simple problem is computed and compared with pre-calculated values |
| 3 | Generation of random numbers | A preset amount of random numbers is generated and printed out |

Continued on next page

3. Implementation Strategy

Table 3.3 – continued from previous page

| Nr. | Feature | Description and expected results |
|-----|---------|----------------------------------|
|-----|---------|----------------------------------|

Table 3.3.: Tests for Integration Application Components

| Nr. | Feature | Description and expected results |
|-----|------------------------------|---|
| 1 | Loading an image | An image is loaded and accessed via the ImageMagick library, properties describing the image, such as pixel number and sizes, are printed out for the verification of the loading |
| 2 | Storing an image | A loaded image is stored with a new name, the image and its copy are compared |
| 3 | Forwarding of the image data | A connection between to pipeline components is established and the data of an image is forwarded, the component receiving the image data stores it and the stored image is compared with the sent image |

Table 3.4.: Tests for Image Processing Pipeline Application Components

3.2.2. System Test

The system test evaluates the correct interactions and the behaviour of the three major components. Furthermore, functions are tested, which proof of correctness is only possible when all components work together. Examples are the simulation of an error during the integration process or the interruption of the image processing pipeline.

The system test requires the correct interaction via RMIX RPC calls and are performed on five nodes. One node is responsible for running the PPM. The other four nodes are used to execute the parallel plug-in components. The next paragraphs describe the system tests.

3. Implementation Strategy

The Monte Carlo integration is performed once with three nodes and once with four nodes for testing the correct and equal distribution of the integration intervals on the available machines. The execution of the integration requires the fact, that the PPM read the input data correctly and loaded the parallel plug-in components.

The test is regarded as successful, if the results collected and added by the PPM are equal to the pre-calculated result of the integration task. Therefore, an example integral is used, which can easily be reproduced.

$$I = \int_{-2}^2 (x^2 - 4) dx \quad (3.1)$$

$$I = \left[\frac{1}{3}x^3 - 4x \right]_{-2}^2 \quad (3.2)$$

$$I = \left(\frac{1}{3}2^3 - 4 \cdot 2 \right) - \left(\frac{1}{3}(-2)^3 - 4 \cdot (-2) \right) \quad (3.3)$$

$$I = -\frac{32}{3} = -10.667 \quad (3.4)$$

The integration is executed on three nodes, whereas the second node is terminated during the computation. So, the PPM is not able anymore to collect the result. The PPM must determine the missed part and resend it to one of the two remaining nodes. The expected results are the notification of an occurred failure, the print out of the missed interval, as well as the resending of this interval and the collection of the result. Last but not least, the correct integration result must be printed out.

For the system test of the image processing pipeline, at least three images are processed. Like for the evaluation of the integration, five nodes are used. One runs the PPM and three a pipeline component, which means that three filters are used. The fifth node is used as a back-up node for restoring a broken pipeline.

In the normal test case, the PPM initialises the pipeline components after reading the configuration terms. Then, it invokes the pipeline. The pipeline units are preset so that the first unit uses a rotation filter, the second an oil painting filter and the last a swirl filter. The correctness of the pipeline can be evaluated by the analysis of the

3. Implementation Strategy

processed images, as the use of the filters in any other order would lead to different results.

After the normal test case, a broken pipeline is simulated. A trigger is integrated into the first parallel plug-in unit. This trigger forces the plug-in to contact the PPM after sending the first image to the second pipeline unit. It informs the PPM that it cannot reach the second component anymore.

Now, it is expected that the PPM reloads the second plug-in component on the fifth node. Furthermore, it must take care that the first and the third unit are informed about the address of the new node which now runs the oil painting filter. Moreover, the first unit must be instructed to resend the already processed images. In the test case the first of the three images is resent. As a reproducible result, all three images are processed and stored by the redirected pipeline.

These introduced test cases are suitable for evaluating the execution of parallel plug-ins at all. Furthermore, they offer possibilities for testing the fault-tolerant mechanisms, which make parallel plug-in applications more secure and flexible. Debug output listings of selected tests can be found in appendix A.2 page 108.

4. Detailed Software Design

4.1. Application Architecture

The parallel plug-in prototype suite was developed in C on a Linux system and implemented in three major modules. These modules included the parallel plug-in manager, the replicated parallel plug-in for Monte Carlo integration, as well as a distributed plug-in for an image processing pipeline. Figure 4.1 gives an overview of the application architecture.

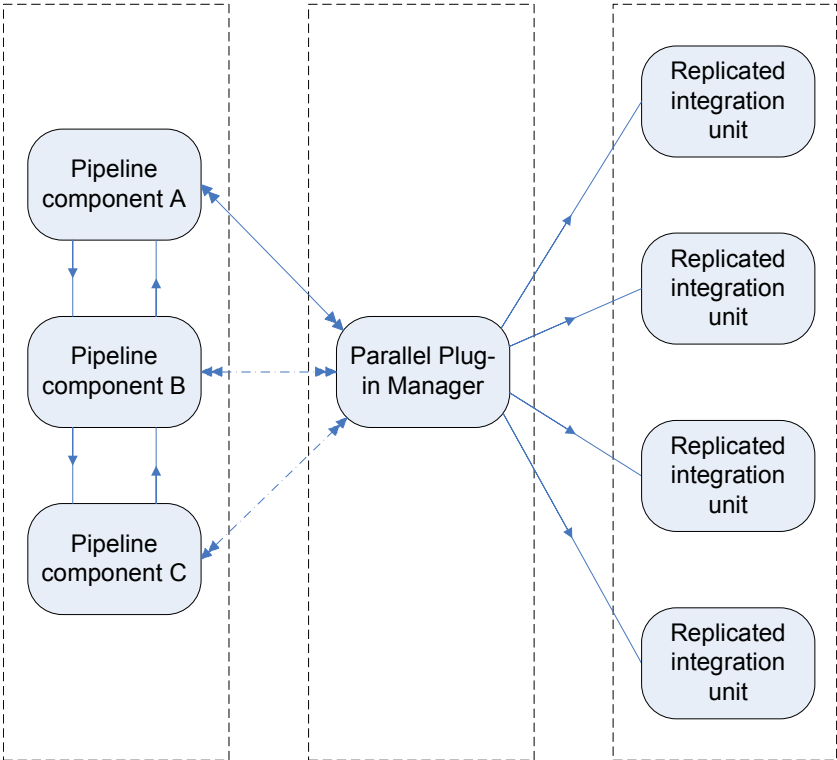


Figure 4.1.: Architecture of the Parallel Plug-in Prototype Suite

The presented modularised design offers flexibility and simplifies the reuse of software. Services are encapsulated in modules together with needed data. This approach facilitates not only the reuse, but also the enhancement or exchange of specific software components.

The central component, illustrated in the figure, is the PPM managing the parallel plug-ins, as well as requesting and providing services. On the left hand side, the distributed parallel plug-in for the image processing pipeline is displayed with three pipeline components. On the right hand side, there is the replicated parallel plug-in for the Monte Carlo integration consisting of for plug-in units all equal to each other and providing the same services.

Besides the components, existing communication capabilities are indicated by arrows, whereas the arrows indicate the direction of the contact establishment. Between the PPM and the parallel plug-in for integral computation, the communication is always initiated by the PPM. It asks the units for their calculation service. Other communication options are not necessary and the replicated plug-in units do not have to be in any contact with each other or ask the PPM for services.

The cooperative work between the distributed parallel plug-in units and the PPM need more communicative interconnections. One-way links between the units are necessary for sending the image data to the next unit, as well as the acknowledgments for successful processing in the opposite direction.

Furthermore, the PPM initialises the parallel plug-in units with information, such as their predecessor or successor plug-in, and it offers them the service for repairing a broken pipe. The connection between the PPM and the first unit of the image processing pipeline is highlighted, because the manager sends a special message to the first unit for invoking the pipeline application.

Figure 4.2 on page 74 presents the major components in more detail. The parallel plug-in manager consists of three parts. The first part is responsible for the management of plug-ins including loading and unloading. This module also includes functions of the external library libConfuse used for reading in configuration files.

The second submodule includes functionalities connected to the execution of the in-

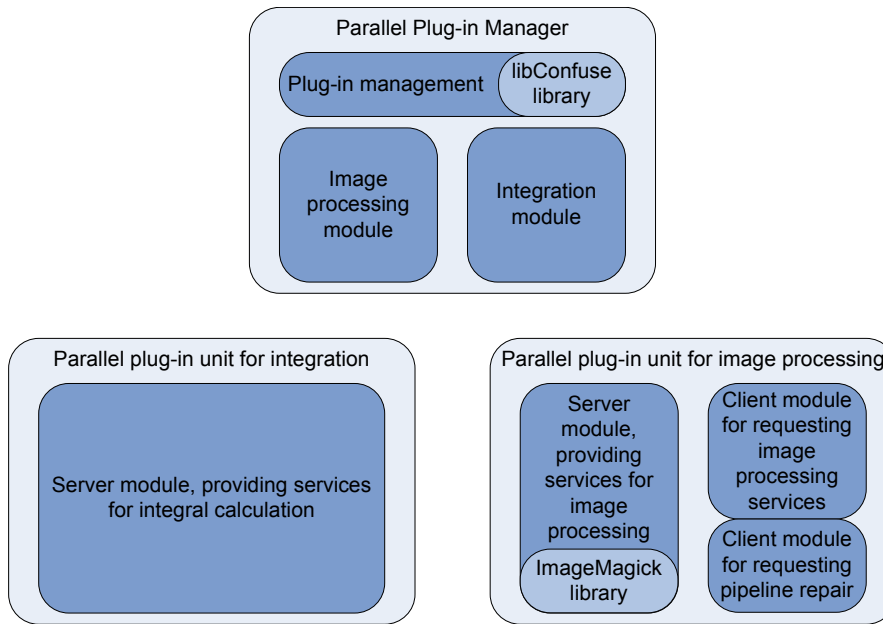


Figure 4.2.: Design of the Prototype Suite Components

tegral computation. Here, client functions are integrated to contact integration units and request their service, as well as the overall control of the integration progress, i.e. the equal distribution of the integration data and the redistribution of missed intervals.

The third submodule is responsible for the initialisation of the distributed parallel plug-in and the invocation of the pipeline. It also provides services for repairing a broken image processing pipeline. The submodules for image processing and integration features are additionally added to the PPM and not essential for its managing tasks. These modules show the extensibility of the PPM beyond its administrative duties.

A parallel plug-in for integration only provides services for integral computations. Here, no further features or external libraries are needed. An image processing pipeline plug-in in contrary includes several submodules. First of all, it has a server part providing features for applying image filters. Therefore, the external ImageMagick library is included.

Additionally, it has two submodules one for requesting services from other pipeline

units, such as forwarding of images, sending of acknowledgments, and requesting pipeline repair services from the parallel plug-in manager. So, the image processing plug-in and the PPM have client and server functionalities.

Nevertheless, all three major components of the prototype suite have the same basis, a plug-in frame consisting of an initialisation and a finalisation function. These functions are executed when the plug-in is loaded or rather unloaded into the Harness runtime environment.

These initialisation and finalisation functions include the counting of loaded instances of the same plug-in in a kernel. Despite the integration, this feature is not used explicitly. It is a control mechanism, which verifies that if a plug-in, for instance the PPM, is loaded twice into the same Harness RTE, it is only executed once.

Moreover, these two functions handle the export of the Harness RMIX plug-in and the plug-in services, which are provided to other parallel plug-in components. As recently as a plug-in exported the RMIX plug-in and its communication interface, other plug-ins are able to contact it and to make use of its services.

4.2. Interface Definitions

The communication is handled via the Harness RMIX library, which provides different possibilities of RPC calls. For using the communication facilities, several preparations have to be met. First of all, the RMIX library must be included into the Harness RTE, which is done by loading the appropriate plug-in.

The loading of the communication plug-in is not yet connected to the start of the Harness RTE. Therefore, the first plug-in loaded into the Harness RTE, which wants to use RPC calls, has to load the RMIX plug-in and export it. The exportation of a plug-in enables the access to the RMIX functions, like sending and receiving calls, as well as the access to preparatory functions like the generation of RMIX specific address structures, including the desired RPC protocol and the hostname.

During the exportation process, an object ID must be specified. This object ID allows the specification or rather the identification of a plug-in's exported interfaces and

methods by a client which wants to access these services. The object IDs must be unique for all the exported plug-ins in one RTE. The following table 4.1 shows the used object IDs in the prototype suite.

| Exported plug-in | Object ID |
|-----------------------------------|-----------|
| Harness RMIX plug-in | 1000 |
| Integration plug-in unit | 1001 |
| Image processing pipeline plug-in | 1002 |
| PPM plug-in | 1003 |

Table 4.1.: Object IDs of the Exported Plug-ins

For enabling the communication, a second important step is the definition of the plug-in interfaces. The interfaces contain detailed descriptions for each integrated RPC communication function. One interface entry consists of the parameters sent with the RPC call, possible return parameters and the server-side and client-side descriptors for the method to be called. This means that the server-side descriptor includes a pointer to the stub function, which accepts the incoming calls.

For each of the three major components of the suite, an interface is defined. The tables 4.2, 4.3 and 4.4 on the next pages contain information about the implemented RPC functions and their parameters. The parameters are presented as input, which is sent to the service provider, and output, which is sent back to the caller. The listings of the interface definitions can be found in appendix A.3.1 on page 120.

| Interface function | Description and parameters |
|--------------------|--|
| Repair pipeline | Service provided to pipeline plug-ins by the PPM to repair a broken pipe - Input[name of the unreachable node (string)] - Output[void] |

Table 4.2.: Interface Definition of the PPM Plug-in

4. Detailed Software Design

| Interface function | Description and parameters |
|----------------------|--|
| Integral computation | Service called for executing an integration on the plug-in unit - Input[lower boundary (double); upper boundary (double); coefficients (double array); quantity of random numbers (integer)] - Output[integration result (double)] |

Table 4.3.: Interface Definition of the Integration Plug-in

| Interface function | Description and parameters |
|---|---|
| Initialising of the pipeline unit | The PPM initialises the pipeline via this function call - Input[image filter applied by the unit (unsigned integer); source directory of the images (string); target directory for the processed images (string); successor pipeline unit (string); predecessor pipeline unit (string); node running the PPM (string)] - Output[void] |
| Invocation of the pipeline process | The PPM invokes the pipeline process by calling the first plug-in unit - Input[void] - Output[void] |
| Resending of the list of processed images | The pipeline plug-in sends its backup list of images to its successor plug-in - Input[void] - Output[void] |
| Availability check | The PPM checks the availability of a loaded plug-in by executing this synchronous RPC call - Input[void] - Output[void] |
| Forwarding an image | The plug-in forwards the specified image to its successor plug-in - Input[image name (string); image data (unsigned char array)] - Output[void] |
| Setting the image counter | The image counter of the plug-in is set to the number of images, which are processed - Input[image counter (unsigned integer)] - Output[void] |
| Update the image counter | In case of a broken pipe the PPM updates the image counter of the reloaded plug-in with the counter of the predecessor plug-in - Input[image counter (unsigned integer)] - Output[void] |

Continued on next page

Table 4.4 – continued from previous page

| Interface function | Description and parameters |
|--|--|
| Update the predecessor plug-in | In case of a broken pipe the PPM updates the predecessor plug-in of the reloaded unit, which gets the value of the predecessor's image counter - Input[name of the host running the reloaded plug-in (string)] - Output[image counter of the predecessor plug-in (unsigned integer)] |
| Update the successor plug-in | In case of a broken pipe the PPM updates the successor plug-in of the reloaded unit - Input[name of the host running the reloaded plug-in (string)] - Output[void] |
| Sending acknowledgment for a processed image | For each stored image an acknowledgment is sent through the pipeline - Input[name of the stored image (string)] - Output[void] |

Table 4.4.: Interface Definition of the Image Processing Pipeline Plug-in

With the help of these definitions, the RMIX plug-in is able to send the information over a network. In the last step, the user is responsible for the implementation of client-side and server-side stubs. The stubs pack and unpack the input and output parameter, whereas the order of the parameter must be the same as in the interface definitions. For each prototype suite component, examples of client-side and server-side stubs can be found in the listings in appendix A.3 page 120.

4.3. Design of Components

4.3.1. Parallel Plug-in Component for Integral Computations

Derived from figure 4.3 on the next page, an integration plug-in component has two main states. After the initialisation process, it remains in a waiting state. The waiting state is left for two different request types, one is the calculation of a certain integral interval and the other option is the termination of the plug-in component.

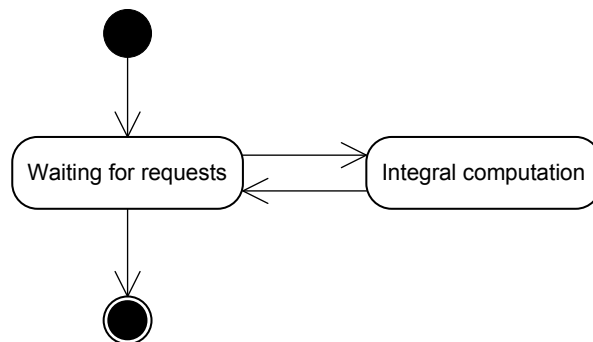


Figure 4.3.: Finite State Diagram for Integration Plug-in Component

For the clear work of the plug-in component, information is stored which must be accessed and processed by different functions. First, the global variables, which are used in the component, are introduced. The global variables generally store management information and data, which must be available during the whole lifetime of the plug-in. As the plug-in remains inactive for the most time and waits for incoming calls, global variables are a possible way for storing this data.

- Control variable in the thread environment, only allowing one thread to be executed in a critical section at a certain point of time
- Counter for the loaded instances of a plug-in, during the project only one instance is loaded pro plug-in
- Identification structure for the exported Harness kernel functions, needed for the correct unexport of the kernel
- Identification structure for the exported integration plug-in functions, needed for the correct unexport of the integration plug-in
- Internal identification for the loaded RMIX plug-in, needed for unloading the plug-in during the termination of the integration application
- The interface descriptions for communication functions necessary for the integration process

In the following, the main functions of a integration unit are introduced. Some of

these functions, i.e. the integral computation are illustrated in more detail for highlighting particular algorithms or their implementation difficulties.

An initialisation function is responsible for checking the number of loaded instances of a plug-in. If it is verified that only one component of the integration plug-in is loaded into the Harness kernel, the initialisation method also prepares the communication capabilities by exporting the RMIX plug-in, the Harness kernel and the integration plug-in itself. Here, the function falls back on the already introduced global variables. After the initialisation process, the parallel plug-in component is in a waiting state.

The opposite of the initialisation is the finalisation process. The implemented finalisation function is called by the Harness RTE, if an instruction for the unloading of the plug-in is received, or the Harness RTE is terminated, which implies the shutdown of all loaded plug-ins. During the finalisation, all exported interfaces and libraries are unexported in reverse order as during the initialisation.

If the exported RMIX library receives an integration request, a new thread is started and the server-side stub is executed. This stub method is responsible for the extraction of the parameters according to the definition of the integration call in the interface description.

After the unpacking of the parameters, the function, which performs the integration, is called. First of all, this function initialises the embedded random number generator of the C library. Then, the integration is performed by implementing the Monte Carlo integration formula (see section 2.5.1 page 33 formula 2.11).

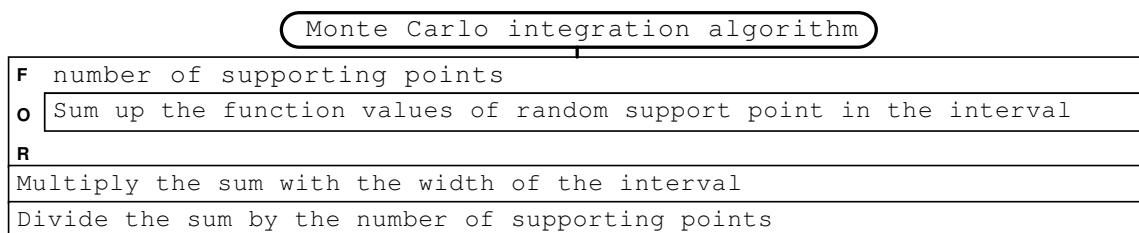


Figure 4.4.: Nassi-Schneidermann Diagram for the Integration Algorithm

For the generation of a supporting point in the interval and the calculation of the function value, two implemented functions are used. After the computation, the result is

returned to the RPC call after packing it into a structure defined by the RMIX integration interface. Although, the plug-in unit is capable of calculating several integrals simultaneously by using the thread approach, a plug-in only executes one integration requests after another.

But, a control mechanism like a mutex is not implemented so that the computation part does not become a critical section. The simultaneous accepting of various integrations is possible and their prevention is incumbent on the calling instance.

4.3.2. Parallel Plug-in Component for Image Processing Pipelines

The software design of the image processing pipeline component is more complex than the design of the integration component. Besides the fact that the pipeline component contains server and client functionalities, it also needs more intelligence concerning the integration of fault tolerance.

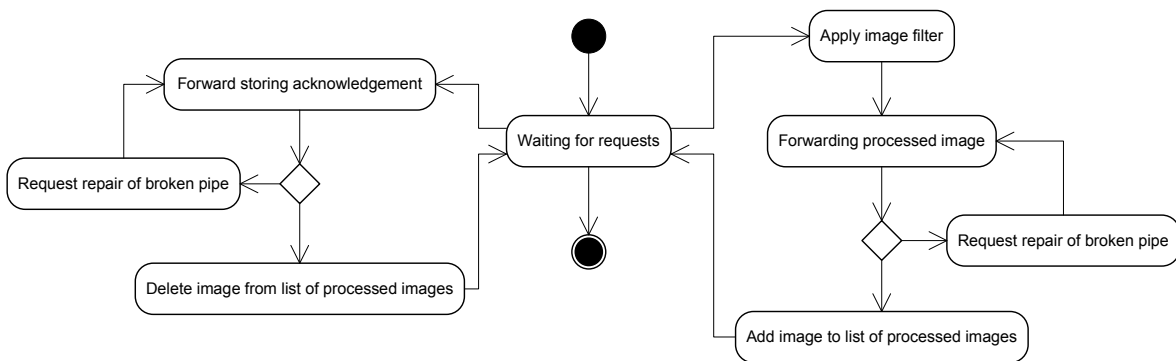


Figure 4.5.: Finite state diagram pipeline component

After the initialisation, a pipeline plug-in is also in a waiting state (see figure 4.5). This state is left for two possible cases, one is the request for an image processing, which includes the application of the image filter, the forwarding of the processed image to the next unit, and the storing of the image data in an internal list for backup purposes.

The second case for leaving the waiting status is the receiving of a storing acknowledgment for one of the processed images. This acknowledgment is forwarded and the according image data is deleted from internal backup list of the pipeline components

which received the notification. As the connection between two pipeline components may be interrupted either in the image forwarding or in the acknowledgment direction, the pipeline component can send a repair request for a broken pipeline to the parallel plug-in manager.

As the image processing problem is integrated more in a peer-to-peer approach, the single parallel plug-in components communicate with each other. Therefore, additional information, than these already introduced in the software design of integration plug-in, must be globally stored. The goal is the implementation of the multiple pipeline model in figure 3.6 at page 64. Therefore, each thread must be able to access the following data in addition to this already mentioned in section 4.3.1 at page 79.

- A variable defining the filter applied by the pipeline component
- Source directory, where the images can be found
- Target directory, where the processed images have to be stored
- The identification of the successor pipeline plug-in for forwarding images
- The identification of the predecessor pipeline plug-in for forwarding acknowledgments
- The identification of the PPM node for requesting pipeline repair
- A counter, counting the images which must still be stored

Regarding the functions, more communication capabilities are realised than in the integration plug-in, but the initialisation and finalisation processes are similar to this design. Therefore, these functions are not described in such detail. For the more detailed description of some functionalities, it is distinguished, whether the component is the first one in the pipeline or not. This is important as some functions are only called in this pipeline unit. First of all, all pipeline units are initialised with values for the mentioned variables above.

After that, the first pipeline unit receives a call to invoke the overall image processing. The plug-in opens the directory and counts the number of images. This number

is sent to all other units in the pipeline so that all image counters are set to the correct value. The sending is carried out via a synchronous call for setting the counter, which internally recalls itself. This is also a check, whether all pipeline units are still available, because the call leaves the first plug-in and only comes back if it successfully reaches the last pipeline unit.

Then, the first plug-in loads one image after another and applies the preset filter. In the case that it has a successor pipeline unit, recognisable on the value of the appropriate global variable, it forwards the image to it. Otherwise, the processed image would be immediately stored. After the successful sending, the image data is inserted into the internal backup list for fault-tolerant purposes. The complete image data is stored so that in a case of a broken pipeline no data get lost. To ensure the consistency of the internal backup list, all accesses are controlled by a mutex and executed in a critical section.

The successor pipeline unit receives a call to its server-side forwarding function. It accepts the image data, applies immediately its filter and also tries to forward the processed data to the next plug-in unit. After forwarding it, the unit also stores the image data with the new applied filter in its own backup list. If the last unit in the pipeline gets the image, it also applies its filter, but it is not able to forward the image. Therefore, it stores the image in the directory, preset during the initialisation process.

After storing the image, the last unit generates an acknowledge message with the name of the stored picture and forwards it to its predecessor unit. The acknowledgment is forwarded from unit to unit until it reaches the first pipeline component. Each component receiving the acknowledgment searches for the image name in its internal backup list and deletes the data. The deletion of entries in the backup list is also executed in a critical section to ensure data consistency.

Figure 4.6 on page 84 gives an overview of the functions for passing an image data set and a storing acknowledgment in a form of Nassi-Schneidermann diagrams. The diagrams include instructions in case of a failed communication attempt. As it is already mentioned, the detection of failures is only possible during communication attempts.

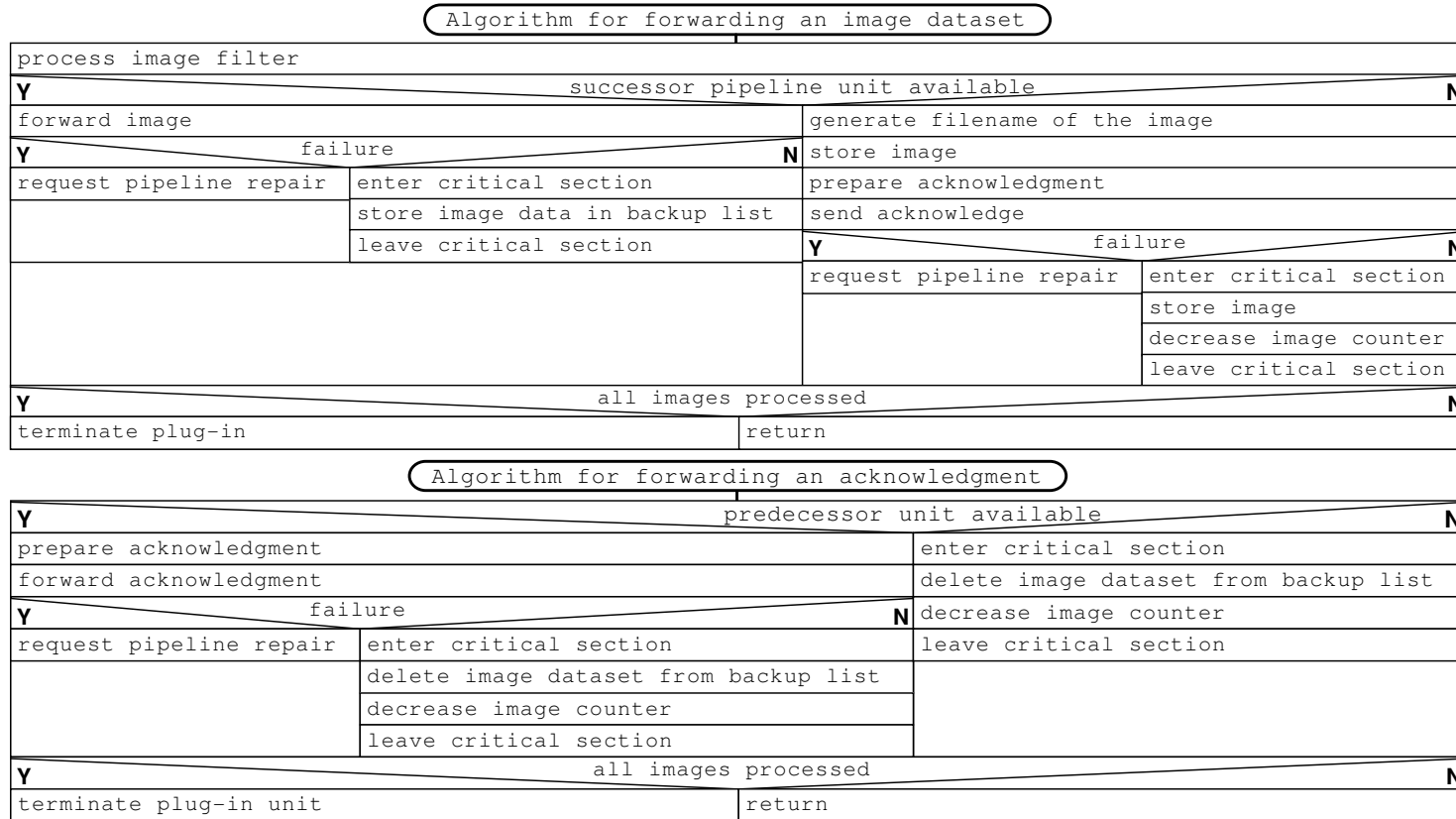


Figure 4.6.: Nassi-Schneidermann Diagrams for Image and Acknowledgment Passing Algorithms

If a failure occurs, the calling pipeline unit requests a pipeline restoration service from the PPM. For instance, if in figure 2.22 at page 46 unit two fails, the first may detect the failure while forwarding an image and the third unit while forwarding an acknowledgment. One of these units or both may now call the parallel plug-in manager. The repair algorithm is explained in detail in the next section.

4.3.3. Service Plug-in for Parallel Plug-in Management

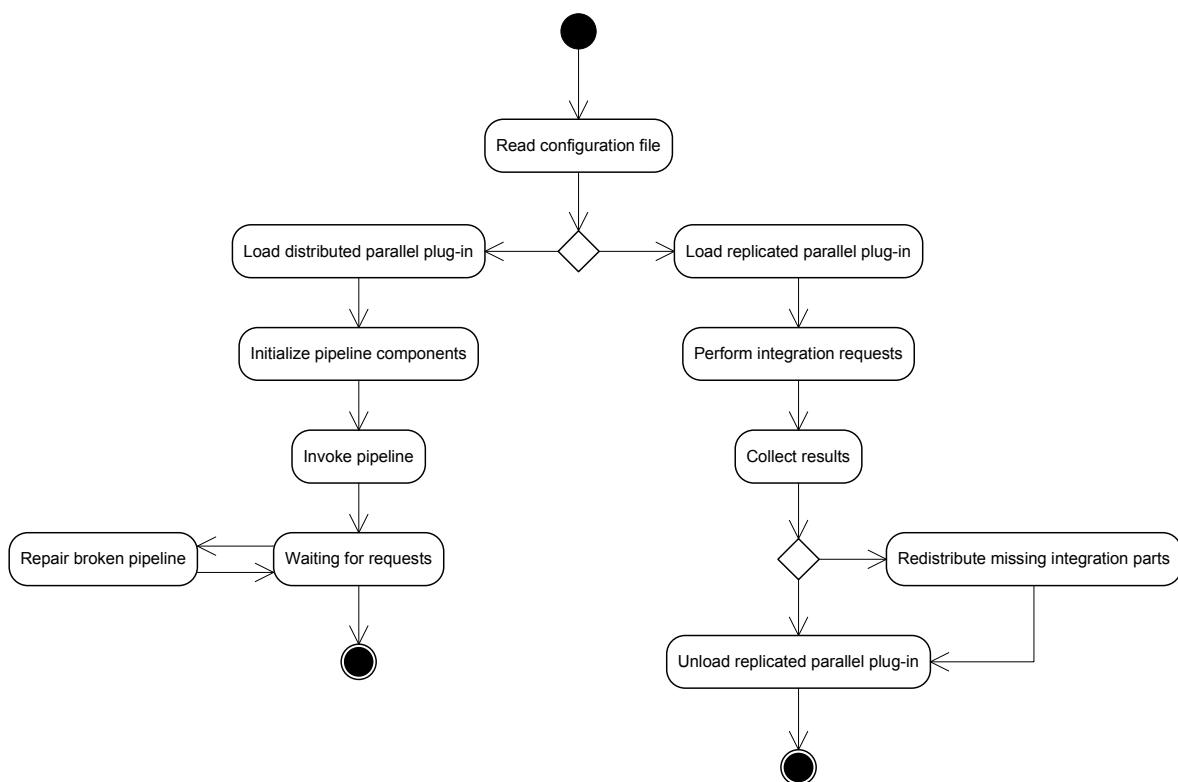


Figure 4.7.: Finite State Diagram of the Parallel Plug-in Manager

The program flow of the parallel plug-in manager depends on how it is configured. Figure 4.7 indicates that the lower complexity in the integration plug-in leads to a higher one for the PPM, which applies inverse for the image processing pipeline.

After the initialisation, the manager reads in the configuration file and loads the desired parallel plug-in. In case of the image processing example, the plug-in units are initialised and afterward the pipeline process is invoked. Then, the PPM remains in

a waiting state and starts a pipeline restoration, if one of the units sends a request referring to this.

Regarding the integration, the complete intelligence is assigned to the PPM, which is responsible for the distribution of the intervals, the checking for failures, and the redistribution of missing parts. Furthermore, it unloads the replicated parallel plug-in after the termination of the computation.

Besides the basic global variables, which are equal to those from the integration plug-in unit, the PPM needs additional information for the management, but also for the cooperative work with the parallel plug-ins. Most of the variables must be stored for fault tolerance in case of a pipeline restoration.

- List with nodes, on which parallel plug-in units are loaded
- List with nodes, which are still available (for the restoration of a broken pipeline)
- The node of the PPM itself (for the restoration of a broken pipeline)
- The names of the distributed plug-in units (for the restoration of a broken pipeline)
- The source directory of the image files (for the restoration of a broken pipeline)
- The target directory for the processed images (for the restoration of a broken pipeline)

Functions have to be realised for the plug-in management, but also for services provided to the parallel plug-ins. Opposite to the pipeline and integration components, the initialisation function of the PPM plug-in loads a function, which starts the management. Otherwise the manager would not perform anything and is captured in a waiting state. The management function is responsible for calling the read method for the configuration file. All functionalities, which affect the configuration file and the conversion of its data into linked lists, are outsourced into an extra utility library.

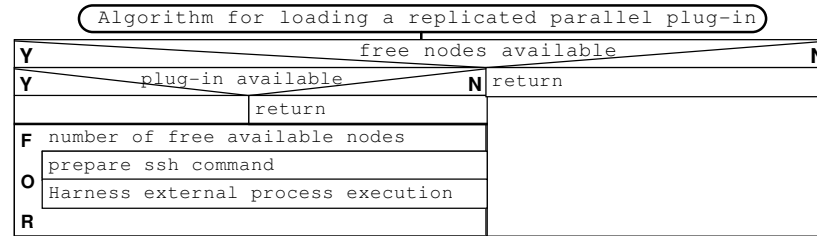
The utility library includes a copy of the configuration file structure, whose content is transferred from the file into the memory via a call of the libConfuse library. Then, the internal configuration structure is converted into lists. For accessing linked lists and the elements they store, the utility collection provides appropriate methods, used by the PPM. After the evaluation of the loaded information, the desired parallel plug-in is started.

Two functions are distinguished, one for loading the replicated parallel plug-in for integration and the second one for loading the distributed pipeline units. The replicated parallel plug-in is loaded on all available machines, but the successful starting of the remote Harness RTE is not verified, which is quite contrary to the loading of the distributed plug-in, where it must be sure that each plug-in unit is loaded. The Nassi-Schneidermann diagrams in figure 4.8 on page 88 show the two different algorithms for loading parallel plug-ins.

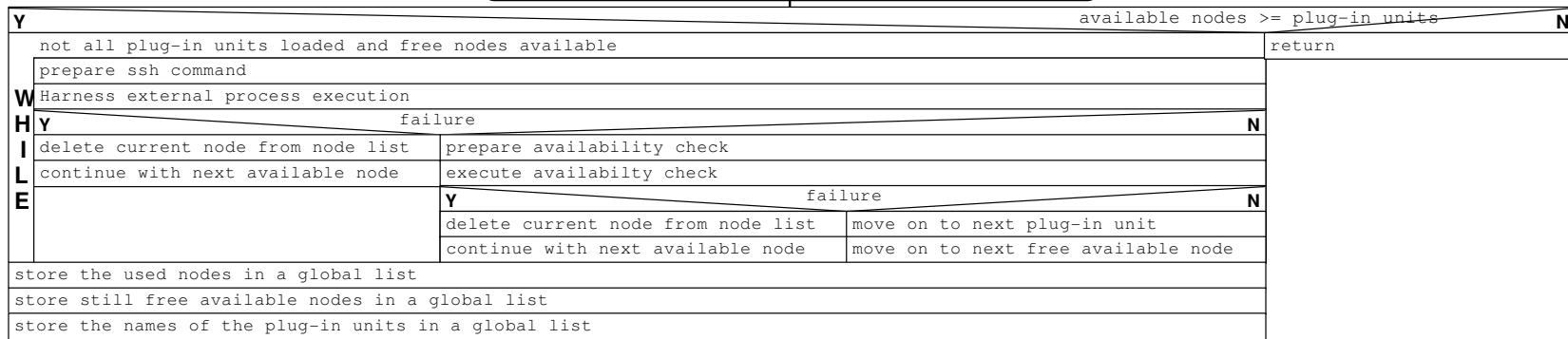
In the case of the image processing application, the PPM now initialises all units and invokes the pipeline by sending a one-way RPC call to the first plug-in unit. During the initialisation process, the PPM must take care of the correct initialisation of all components, especially the first and the last one in the pipeline. The first pipeline unit has no predecessor but needs the source directory with the images and the last unit has no successor but needs the target directory for the images.

As the pipeline components perform the image processing self-governed, the PPM plug-in is in a waiting state providing the pipeline restoration service. If it receives such a request from a plug-in, several instructions are carried out. On page 88 figure 4.8 gives an overview of the algorithm.

First, the whole restoration process is handled in a critical section, which is necessary as the predecessor and the successor plug-in components may start a repair request, and because of the Harness RTE thread environment, two threads may try to fix the problem simultaneously. In the critical section, the failed node is searched in the list of used nodes. If it cannot be found, it is assumed that another plug-in unit requested a repair before and the problem is solved, otherwise the restoration is started.



Algorithm for loading distributed parallel plug-in



Algorithm for pipeline restoration

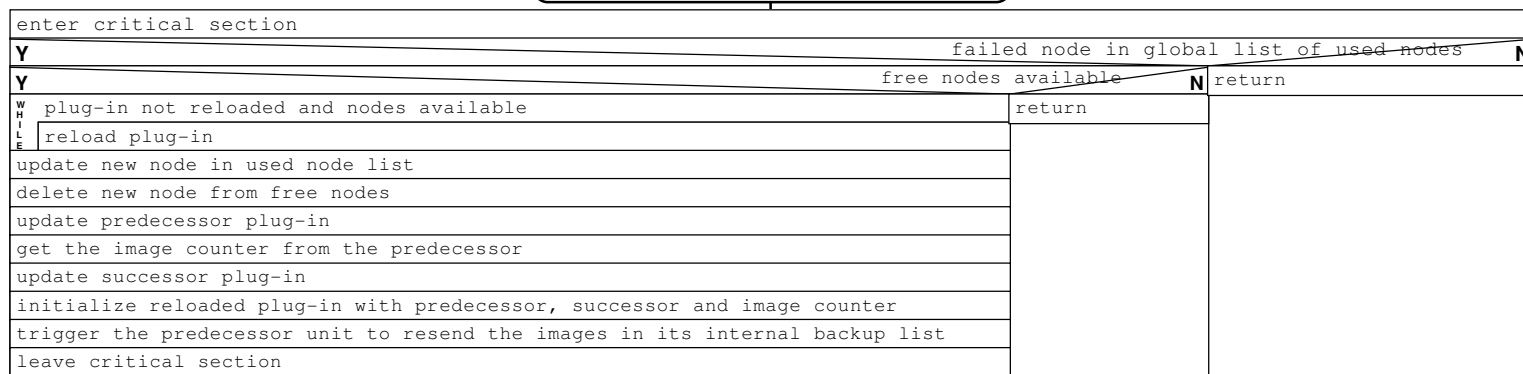


Figure 4.8.: Nassi-Schneidermann Diagrams for Parallel Plug-in Loading Algorithms and Pipeline Restoration

If there are still available nodes, the failed plug-in is reloaded on one of these. If there is no free node left, the pipeline cannot be repaired. After reloading the plug-in, the predecessor and the successor plug-ins are updated with the contact information of their new neighbour. Furthermore, the image counter from the predecessor plug-in is copied to the reloaded one. So, both pipeline components are at the same state regarding the processed images.

Then, the reloaded pipeline component is initialised and ready for operation. The last issue concerns the possible loss of images, which were just processed by the failed pipeline component. Therefore, the predecessor plug-in is triggered to resend its internal backup list with the stored images. If the first unit of the pipeline is reloaded, the entire pipeline is just reinvoked by the PPM, and all images are reprocessed. At the end of the restoration process, the PPM leaves the critical section and the pipeline is repaired.

Regarding figure 4.6 at page 84, the internal backup lists are updated after the successful forwarding of an image or an acknowledgment. This is due to the fact that, the failure detection takes place while the message is sent. If a plug-in has been reloaded, the pending message is immediately resent to the new loaded pipeline component. Therefore, it does not have to be in the backup list yet, otherwise it would be sent twice.

Regarding the integration problem, the PPM covers the client part. An algorithm is implemented, which sends integral parts to the available units. This algorithm is described in figure 4.9 page 90. The integration dataset is equally distributed over the available nodes, whereas a counter is increased for occurring errors and the ranks of failed nodes are stored. The rank is used for the redistribution of the missed part, as it is possible to recalculate the missed interval.

If failures occurred and at least one node is still available, the missing part(s) are redistributed. Therefore, an additional redistribution algorithm is implemented, which is executed if one or more failures occurred. The ranks, which indicate the missed integration parts, are stored in an extra array. As long as there are available nodes the application tries to recalculate all failed integral intervals.

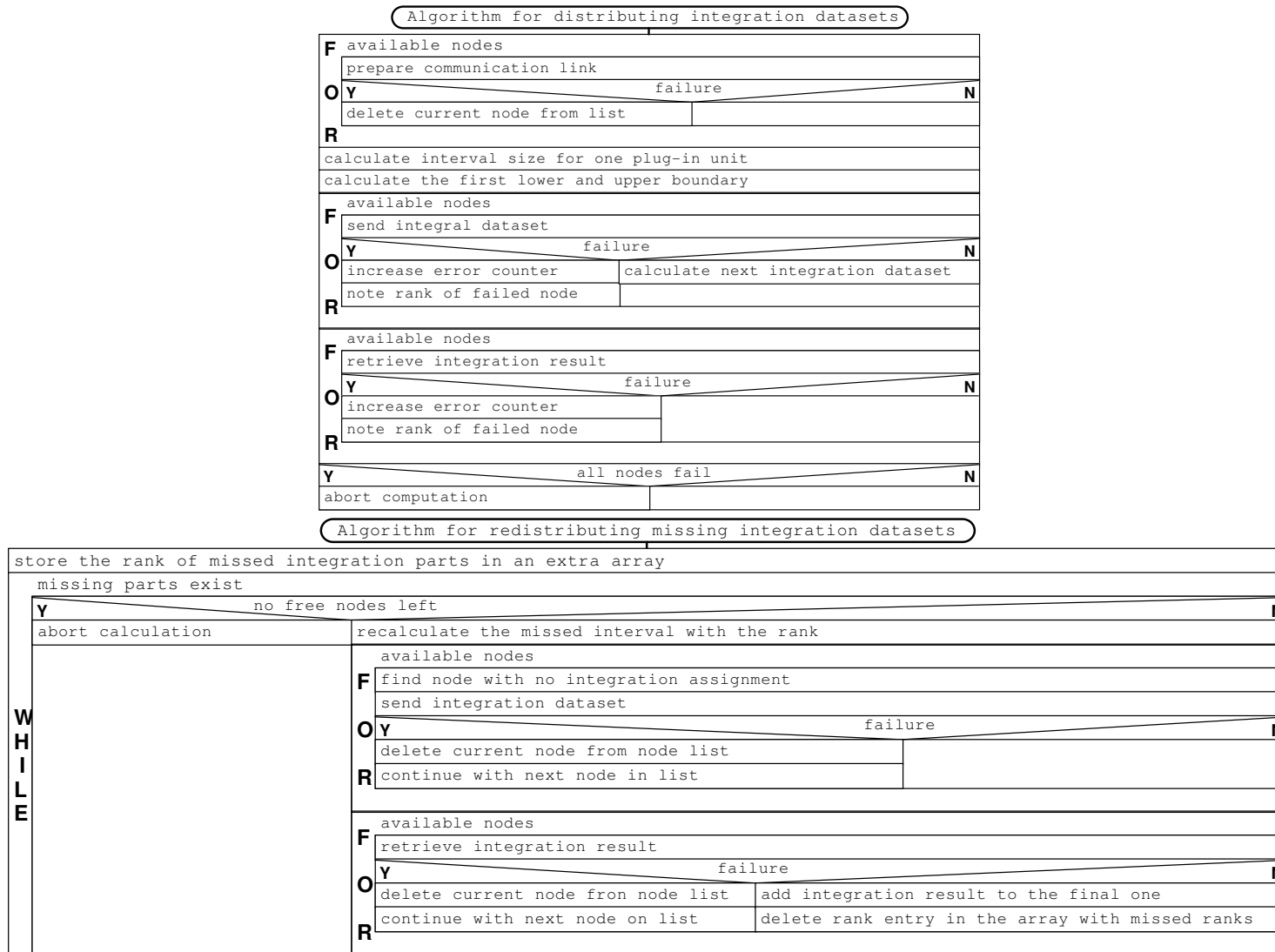


Figure 4.9.: Nassi-Schneidermann Diagram for Integration Distribution and Redistribution Algorithms

First, the failed interval is recalculated according to the stored rank in the array. This dataset is sent to an available node. If there is more than one interval missing, the next missed interval is forwarded to a node, which does not have an integration order yet. So, a form of load balancing is also included. An easier way would be to send all missed parts to the same integration unit, which excludes load balancing.

If the sending to an integration service failed, the node, providing the service, is delegated from the list of available nodes and the dataset is forwarded to another node. After all missed datasets are sent, the collection of results is started. The nodes, which got an integration order, are contacted and if one failed, it is also deleted from the list of available nodes and the missed integration part is not marked as solved.

But if a result is received, the missed part is marked as solved by deleting its entry in the array with the stored ranks. The result itself is added to the already computed integration results. Thus, the integration problem can mostly be solved, as it is unlikely that all nodes will fail simultaneously if a great amount of resources is available.

5. Conclusion

5.1. Results

The major goal of this Master thesis project was the evaluation of parallel plug-in technologies, as well as the gaining of first experiences with them. Therefore, different concepts were introduced. One was the replicated parallel plug-in and the other one the distributed parallel plug-in. Furthermore, an additional plug-in concept was designed to provide services for parallel plug-ins. These models covered the main use cases of applications for the Harness system.

The basic concept of parallel plug-ins was investigated, which meant the combination of the advantages of the plug-in technology and the advantages of a distributed runtime environment, such as the Harness RTE. The two examples, which provided various facets of parallel plug-in technology, familiarise the user with the possibilities of Harness and parallel plug-in applications. In addition to this, the concepts help the users and programmers simplify their work. Solutions were presented, which dealt with the three major problems or features of parallel plug-ins.

- (un)loading of parallel plug-ins
- inter plug-in communication
- failure tolerant mechanisms

The introduction of these three basic features was connected with the reuse of plug-in bodies or certain functions. The code reuse was already taken into consideration during the system design and the derivation of implementation strategies. For instance,

5. Conclusion

the configuration file combined with the read in function can already contain the information for several parallel plug-ins, although this feature is not yet used. But it opens possibilities to load many parallel plug-ins with one loader. Therefore, third party libraries were also included, which facilitate the process of the configuration adaptation.

Besides the obvious features of (un)loading plug-ins, the PPM can also be used to perform computation preparations, as shown in the Monte Carlo integration example. According to a template, the plug-in frames can be used and desired changes can be carried out. Especially, the existing integration prototype may be extended with additional integration algorithms or random number generators.

The implementation of the image processing pipeline also showed the possibility of parallelising processes, which were actually sequential. Especially this use case highlighted the communication capabilities of the Harness RMIX library, responsible for the integration of RPC calls in the Harness RTE. The available functions including asynchronous and synchronous RPC were used with their various options. The developed system designs emphasized these features of the Harness runtime environment.

In addition to the communication capabilities, fault tolerance mechanisms were presented and included in the system design. These mechanisms included partial success during the loading of plug-ins and simple fault recovery. The prototypes were able to perform their tasks even if some of the participating parallel plug-in units failed. This is a new feature compared to PVM or MPI, where a failed node often can be equated with the abortion of a whole program.

In the scientific world, regarding problems like the genome research, application have to meet certain requirements. Especially the scalability and reliability are very important. Scientific problems often deal with huge amounts of data and sometimes have to run for months without interruptions. The designed, implemented and tested prototypes fulfil these requirements.

The replicated parallel plug-in for integration shows a solution option for a scalable problem and both parallel plug-in prototypes include a higher reliability even if components fail. But it is elemental that there are still available nodes where a component

5. Conclusion

can be reloaded.

Regarding the project progress, the major goals were achieved and the implementations of the basic features of the replicated and distributed parallel plug-ins, as well as the fault tolerance mechanisms were completed. Besides these two objectives, the Parallel Plug-in Manager was realised in an easy, adaptive way. The reuse of ideas, implementation options and software components is encouraged.

The components tests and system tests were successful executed. Each component function mentioned in section 3.2.1 at page 65 was implemented and tested. The test logs in appendix A.2 page 108 show the process of the system tests, which were also introduced in the testing strategies section 3.2.2 page 69. After the intended crash of a parallel plug-in unit, both applications were able to finish their tasks successful.

The gained experiences show that the plug-in technology was successfully joined with the Harness RTE. Due to the parallel plug-in manager, it was much easier to handle entire parallel plug-ins on a set of nodes. Depending on the application, it was easy to build a plug-in frame which can be integrated into the RTE. Often, it is possible to copy the basic plug-in frame and then add the desired functionalities.

The implementations of the algorithms for the integral computation and the execution of the image processing pipeline were more difficult. The algorithms with their basic features were not a problem. A challenge was the realisation of fault tolerance. The use of RMIX allowed the detection of failed components only during communication attempts.

Fault-tolerant mechanisms were also traded against memory space, regarding the backup lists of images in each pipeline unit, the higher intelligence and algorithm complexity of one (master process of the parallel plug-in for integration) or all units (image processing pipeline), and higher communication costs, for instance, the sending of acknowledgments within the pipeline and the update process after the breakdown of a pipeline unit.

Parallel plug-ins are a possibility to implement scientific applications in a parallel manner. The basic functions of an algorithm can be easily realised. Code reuse in form of plug-in frames is facilitated. The communication and fault tolerance make

higher demands on the user, especially the interface definitions and the intelligence for handling a failure. Therefore, the example applications are a chance to start into the world of parallel plug-ins. In appendix A.3 page 120 the source listings can be found, which provide insight into the functionalities of parallel plug-ins.

5.2. Future Work

The chosen use cases can still be improved and extended. As already mentioned the replicated parallel plug-in for Monte Carlo Integration may be expanded to a full utility package, covering different mathematical computations which can be implemented in parallel.

The parallel plug-in may be integrated in other scientific applications, which take advantages of such a mathematical collection. One point, which has to be improved first regarding the Monte Carlo Integration, is the implementation of a more appropriate pseudo random number generator. Furthermore, algorithms may be included, which evaluate the problem size and then load additional plug-in components into the parallel plug-in for a higher level of parallelism.

Regarding the image processing pipeline parallel plug-in, the fault-tolerant mechanisms can still be improved. The problem that a failure is only detectable during communication activities exists so that in the current implementation a failure may be undetected. In the unlikely case that one of the pipeline processes is very slow in comparison to the other units or the work load is not balanced. This unit may still work but all other units are in a waiting process.

According to the image processing problem, one filter takes a lot of time and all other units do not work as there is a kind of data jam. Therefore, no images and/or acknowledgments are sent through the pipeline. If the only working unit brakes down, the neighbours would not detect that. Now, various options can be applied to solve this problem. All of them deal with the exploitation of the Harness thread environment for implementing a polling to check the availability of neighbours, whereas the pooling is performed by separate threads.

5. Conclusion

Research in parallel plug-ins can also be combined with the research in other new areas of computer engineering. One combined research project may be the management of parallel plug-ins on multi-core processors. Multi-core processors contain two or more fully equipped processors, which improves the performance of microprocessors, i.e. regarding pipelining.

For instance, research is possible in the internal plug-in communication where the plug-in components run on the same node but on different cores. In this case also the loading of parallel plug-ins has also to be reconsidered.

This Master thesis project provides interested programmers and scientists with a new technology to implement their problems in parallel, but it also introduces the basis for a lots of additional research.

References

- [BDF⁺98] Micah Beck, Jack J. Dongarra, Graham E. Fagg, G. Al Geist, Paul Gray, James Kohl, Mauro Migliardi, Keith Moore, Terry Moore, Philip Papadopoulos, Stephen L. Scott, and Vaidy Sunderam. *HARNESSE: A Next Generation Distributed Virtual Machine*. Elsevier Preprint, June 1998.
- [Beo05] Beowulf.org. Beowulf project overview. <http://beowulf.org/overview/index.html>, 2005. [Online; accessed 14-January-2006].
- [Buy99] Rajkumar Buyya. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall PTR, NJ, USA, 1999.
- [Chi] Tan Chee Chiang. High performance computing: Current trends and technology. http://www.nus.edu.sg/comcen/svu/publications/SVULink/vol_1_iss_1/hpc-trends.html. [Online; accessed 29-September-2005].
- [EG05a] C. Engelmann and G. Geist. *A Lightweight Kernel for the Harness Meta-computing Framework*. 2005.
- [EG05b] C. Engelmann and G. Geist. *RMIX: A Dynamic, Heterogeneous, Reconfigurable Communication Framework*. 2005.
- [Eng] C. Engelmann. Harness program package. <http://www.csm.ornl.gov/~engelman/>. [Online; accessed 02-February-2006].
- [ES05] C. Engelmann and S. L. Scott. Concepts for high availability in scientific high-end computing. In *Proceedings of High Availability and Performance Computing Workshop (HAPCW) 2005*, Santa Fe, NM, USA, October 2005.

References

- [FGB⁺04] Graham E. Fagg, Edgar Gabriel, George Bosilca, Thara Angskun, Zhizhong Chen, Jelena Pjesivac-Grbovic, Kevin London, and Jack J. Dongarra. Extending the mpi specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference (ICS)*, 2004.
- [fHEC] The Centre for High-End Computing. The centre for high-end computing - introduction and background. <http://chec.it.nuigalway.ie/>. [Online; accessed 13-January-2006].
- [Fou05] Free Software Foundation. autoconf. <http://www.gnu.org/software/autoconf/>, 2005. [Online; accessed 15-February-2006].
- [GBD⁺94] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.
- [GL] William Gropp and Ewing Lusk. Pvm and mpi are completely different. <http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html>. [Online; accessed 14-January-2006].
- [Gro04] William Gropp. Tutorial on mpi: The message-passing interface. *Future Generation Computing Systems*, September 2004. [Online; accessed 14-January-2006].
- [Har05] Harness. *HARNNESS: Reconfigurable Distributed Environments for Heterogeneous Metacomputing*. 2005.
- [Has00] M. Hasenbusch. Monte carlo simulations in statistical physics. <http://linde.physik.hu-berlin.de/~hasenbus/vorlesung.html>, 2000. lecture notes HU Berlin, Germany.
- [Heb99] Shane Hebert. Message passing interface (mpi) faq. <http://www.cs.uu.nl/wais/html/na-dir/mpi-faq.html>, May 1999. [Online; accessed 14-January-2006].
- [Hed04] Martin Hedenfalk. libconfuse. <http://www.nongnu.org/confuse/>, 2004. [Online; accessed 13-January-2006].

References

- [Hei05] Harald Heim. Plugin essentials. <http://thepluginsite.com/knowhow/tutorials/introduction/introduction.htm>, 2005. [Online; accessed 20-February-2006].
- [Her04] Helmut Herold. *Linux/Unix Systemprogrammierung*. Addison-Wesley, September 2004.
- [Hol97] Heiko Holtkamp. Introduction in tcp/ip, June 1997. Universität Bielefeld.
- [HP02] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann, June 2002.
- [Kom05] Martin Kompf. Zufallszahlen. <http://cplus.kompf.de/artikel/random.html>, 2005. [Online; accessed 10-November-2005].
- [LLC05] ImageMagick Studio LLC. Imagemagick - image processing library. <http://www.imagemagick.org/script/magick-wand.php>, 2005. [Online; accessed 13-January-2006].
- [MKL⁺02] Dejan S. Milojcic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja1, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. *Peer-to-Peer Computing*. HP Laboratories Palo Alto, March 2002.
- [MMS02] Johannes Mayer, Ingo Melzer, and Franz Schweigert. *Lightweight Plugin-Based Application Development*. University of Ulm, Ulm, Germany, October 2002.
- [MR03] Prof. J. Mnich and Dr. S. Roth. Zufallszahlen und monte-carlo-methoden. <http://www.physik.rwth-aachen.de/~roth/eteilchen/MonteCarlo.pdf>, 2003. [Online; accessed 10-November-2005].
- [oT] Innovative Computing Laboratory University of Tennessee. Harness. <http://icl.cs.utk.edu/harness/>, note = [Online; accessed 15-October-2005].
- [otPotUS04] Execution Office of the President of the United States. Federal plan for high-end computing, May 2004. Report of the High-End Computing Revitalization Task Force (HECRTF).

References

- [Sch03] Prof. Wolfgang Schebesta. Data and compute networks, April 2003. lecture notes.
- [SL] Computer Science and Mathematics Division Oak Ridge National Laboratory. Harness the power of the network. <http://www.csm.ornl.gov/harness/>. [Online; accessed 15-October-2005].
- [SL05] Computer Science and Mathematics Division Oak Ridge National Laboratory. Pvm private virtual machine. <http://www.csm.ornl.gov/pvm/>, December 2005. [Online; accessed 14-January-2006].
- [SOHL⁺96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.
- [Uni] Emory University. Harness...the power of the network. <http://www.mathcs.emory.edu/harness/>, note = [Online; accessed 15-October-2005].
- [Wik06a] Wikipedia. Fault-tolerant design. http://en.wikipedia.org/wiki/Fault_tolerant_design, January 2006. [Online; accessed 18-January-2006].
- [Wik06b] Wikipedia. Fault-tolerant system. http://en.wikipedia.org/wiki/Fault-tolerant_system, January 2006. [Online; accessed 18-January-2006].
- [Wik06c] Wikipedia. Network topology. http://en.wikipedia.org/wiki/Network_topology, January 2006. [Online; accessed 18-January-2006].

A. Appendix

A.1. Program Manual

A.1.1. Installation

An requirement for the installation of the parallel plug-in prototype suite is the availability of the Harness runtime environment including the RMIX library, as well as the libConfuse and ImageMagick libraries. Appropriate instructions for the installation of these packages can be found at their source pages [Eng][Hed04][LLC05].

The parallel plug-in suite is merged to package, including an autotools environment. This environment consists of the following directory structure.

| Directory | Description |
|-----------------|--|
| data | contains the configuration files for the components of the prototype suite |
| imageprocessing | contains the sources of the parallel plug-in for an image processing pipeline |
| images | contains three example images |
| imagetarget | contains the processed versions of the example images |
| include | contains the RMIX interface definitions of the three components of prototype suite |
| montecarlo | contains the sources of the parallel plug-in for integral computation |
| ppm | contains the sources of the parallel plug-in manager and the additional utility library for reading in configuration files |

Table A.1.: Directory Structure of the Prototype Suite Implementation

As parallel plug-in software package uses the features of the autotools, the instal-

lation follows the generic instructions provided by the Free Software Foundation [Fou05]. The following paragraphs are extracted from the installation advices included in the autotools.

The main directory contains a `configure` script, which handles the compilation and installation process. It attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create files, such as Makefiles and header files. Finally, it creates a shell script `'config.status'` that you can run in the future to recreate the current configuration, and a file `config.log` containing compiler output (useful mainly for debugging `configure`).

The file `configure.ac` is used to create `'configure'` using `'autoconf'`. Only the `configure.ac` is needed if `'configure'` has to be changed or regenerated by using a newer version of `autoconf`. The file `configure.ac` is also used by `'automake'` to generate `Makefile.in` files from `Makefile.am` files. So that `'configure'` is able to create the final Makefiles from the `Makefile.in` files.

Other tools, such as `'autoheader'`, `'gettext'` and `'aclocal'`, also use the file `configure.ac` to extract information for automatically creating header files, adding foreign message support to applications and supplying M4 macros to `'configure'`. Additionally, the program `'libtool'` provides an extensive support for building static and shared libraries with `'automake'` template `'.am'` files.

To simplify the use of the autotools (`'autoconf'`, `'automake'`, `'autoheader'`, `'gettext'`, `'libtool'` and `'aclocal'`) this distribution supplies an `'autogen.sh'` shell script. It scans `configure.ac` for specific macro calls related to the autotools, checks which of them are really needed, checks for their existence and executes them in the correct sequence.

Furthermore, `'autogen.sh'` finds and processes all `configure.ac` files in all sub-directories recursively and calls the top-level directory `'configure'` with the arguments supplied to `'autogen.sh'`. So that `'autogen.sh'` recursively creates all needed `'.in'` files and `'configure'` shell scripts, and configures the top-level package and all sub-packages where recursive `'configure'` is used (see `'autoconf'` manual).

The source distribution is released after executing `'autogen.sh'`, so that it contains all necessary files prepared to be modified by `'configure'` on the target system. The

'autogen.sh' shell script may only be used after changing configure.ac or Makefile.am files during the development of the software package.

Some systems require unusual options for compilation or linking that the 'configure' shell script does not know about. Run 'configure --help' for details on some of the pertinent environment variables. You can give initial values for configuration parameters by setting variables in the command line or in the environment. By default, 'make install' will install the package's files in '/usr/local/bin', '/usr/local/man', etc.

For the installation process, the following steps have to be proceeded. The source distribution has to be unpacked into a temporary directory. Then it has to be configured, compiled and installed as root into the system directory tree.

Before the compilation process, the line 103 in the file readconf.h found in the subdirectory "parallelplugins/ppm/libplutils" has to be edited. This line defines the directory, where the configuration file for the PPM can be found. As it is not possible to add a parameter to a plug-in, which will be loaded, the plug-in has to search for a configuration file after the start.

- cd /tmp
- tar -xzf parallelplugins.tar.gz
- edit the path in parallelplugins/ppm/libplutils/readconf.h line 103
- cd parallelplug-ins
- ./configure
- make all
- make install
- cd ..
- rm -f parallelplugins.tar.gz
- rm -fr parallelplugins

If tar does not accept the 'z' option please use instead:

- cd /tmp
- gunzip parallelplugins.tar.gz
- tar -xf parallelplugins.tar
- edit the path in parallelplugins/ppm/libplutils/readconf.h line 103
- cd parallelplug-ins
- ./configure
- make all
- make install
- cd ..
- rm -f parallelplugins.tar.gz
- rm -fr parallelplugins

A.1.2. Example Configuration Files

A.1.2.1. Configuration File for the Parallel Plug-in Manager

```
1 # this is the configuration file for the PPM
3 nodelist
4 {
5     addresses = { "h1", "martha", "h2", "h3" }
6 }
7 replicated
8 {
9     plugins = {libintegral.0.0.0.so}
10 }
11
12 distributed
13 {
14     plugins = { "libimgproc.0.0.0.so",
15                "libimgproc.0.0.0.so",
16                "libimgproc.0.0.0.so"
17            }
18 }
19 }
```

A. Appendix

```
21 input
   {
23     files = {"/home/ronald/parallelplugins/data/montecarlo.input",
               "/home/ronald/parallelplugins/data/image.input"}
25 }

27 mode = 2

29 ppmnode = "kid"
```

This file is read in via the libConfuse interface. It consists of different sections and options. In line three, the node section starts. All nodes are listed, which are available to run a component of a parallel plug-in. The second section is for the replicated parallel plug-in, beginning at line eight. In the case of the prototype suite, the name of the replicated Monte Carlo Integration plug-in is recorded in form of the library name.

The next section, starting at line 13, contains a list of components belonging to a distributed parallel plug-in. In case of the prototype suite, all the units are the same library due to the implementation strategy. The number of available nodes must be at least equal or higher as the number of distributed plug-in units. Each unit will be loaded on a separate node.

The input section at line 21 includes files, which contain inputs for parallel plug-ins. In the example, the first file is for the integration parallel plug-in and the second file for the image processing parallel plug-in. After the input section, two options follow.

The first option mode distinguishes whether the PPM will load the mentioned replicated parallel plug-in (mode equals one) or the distributed parallel plug-in (mode equals two). The last option ppmnode indicates the node, which runs the Parallel Plug-in Manager.

A.1.2.2. Configuration File for the Integral Parallel Plug-in

```
1 10000
  -2 2
3 10
  -4 0 1 0 0 0 0 0 0 0
```

This file has a simple design. Line one contains the number of supporting points, which will be generated randomly. The next line indicates the lower and upper inte-

gration boundaries. In line three the number of coefficients is set. These coefficients express the function as a polynomial. In line four, the coefficients are written, beginning with the smallest coefficient.

$$\int_a^b f(x)dx \quad (\text{A.1})$$

$$\int_{-2}^2 (0x^9 + 0x^8 + 0x^7 + 0x^6 + 0x^5 + 0x^4 + 0x^3 + 1x^2 + 0x^1 - 4x^0)dx \quad (\text{A.2})$$

$$\int_{-2}^2 (x^2 - 4)dx \quad (\text{A.3})$$

A.1.2.3. Configuration File for the Image Processing Parallel Plug-in

```
# this is the configuration file for the image processing pipeline
2 sourcedir = "/home/ronald/parallelplugins/images"
4 targetdir = "/home/ronald/parallelplugins/imagetarget"
```

This input file is read in via the libConfuse interface. It contains two entries. The first entry specifies the the source directory, in which the first pipeline unit will find the images, and the second entry defines the target directory, in which all processed images will be stored.

The program assumes that all files, which can be found in the source directory, are images. There is no verification performed while reading in possible image files. Furthermore, it is not checked, whether an image, which is stored, overwrites an older file with the same name.

A.1.3. Program Execution

After the successful installation of the program package. The configuration files have to be edited. Examples for configuration files and explanations regarding the options can be found in appendix A.1.2 page 104. If all adjustments were performed, the Parallel Plug-in Manager can be started.

There are two possibilities for starting the PPM:

- `harnessd -l libppm.0.0.0.so`
- `harnessd.debug -l libppm.0.0.0.so`

Both options start a Harness daemon, which initialises the Harness runtime environment and immediately loads the Parallel Plug-in Manager, whereas the dynamic link library, which includes the PPM, is integrated into the Harness RTE.

The difference between these two start possibilities is the additional output of debug and standard information by using the second one. This output is printed to the console, in which the PPM is started and run. Depending on the configuration of the Parallel Plug-in Manager, it will load the replicated integration plug-in or the image processing pipeline.

It is recommended, that the user starts the program suite with the second option to see all information outputs. Some example outputs can be found on page 108 in appendix A.2. Now, it is possible to adjust the configuration files, especially the one for PPM, to generate different initial situations, i.e. adding some names to the node list, which cannot be resolved. This offers possibilities to test the failure handling of the PPM loading functions.

For testing the failure handling mechanisms of both use cases, it is possible to disconnect or kill a node, especially the Harness runtime environment, while the integration or image processing take place. By adjusting the file "`paralleplugins/imageprocessing/libimgproc/imgproc.c`", it is possible to force a pipeline restoration. In line 1537 of that file, the uncomment section contains a trigger, which forces the last unit of the pipeline to contact the Parallel Plug-in Manager for a pipeline restoration after generating the acknowledgment for the first processed image. The pipeline unit informs the PPM that it cannot reach its predecessor anymore.

If there is an available node, the PPM will install a pipeline redirection for the unit before the last unit of the pipeline. Furthermore, in line 3229 is a second uncomment trigger. This trigger causes the first pipeline unit to contact the PPM for a restoration after sending the first image to the successor unit. Both trigger are uncomment to prevent an interruption of the normal processing.

For testing the integration redistribution, it is advisable to disconnect one of the calculating plug-ins, while computing a bigger problem, otherwise it is necessary to slow down the computation artificially, i.e. by adding a sleep command on fast computers. In file "parallelplugins/montecarlo/libintegral/integral.c" line 654, a uncomment section can be found, which executes a simple sleep command so that the user has a greater time frame for terminating one of the computation units.

A.2. Program Output Listings

The presented listings are extracted from generated log files. Three dots indicate that the listing is shortened at this place. The full output listing can be found on the enclosed data media. A standard log file entry consists of "info:library name:process id:source file:line in the source file:function name:message".

A.2.1. Output Listings of the Parallel Plug-in for Integration

A.2.1.1. Performed on Four Nodes

```
1 ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
2 ...
3 info:libppm:1031:ppm.c:231:ppm_init:libppm is starting
  info:libppm:1031:ppm.c:436:ppm_init_rmix:start initialization of RMX
4 ...
5 ...
  info:libplutils:1031:readconf.c:114:configuration_read_file:start reading configuration file
6 ...
7 info:libplutils:1031:readconf.c:388:configlist_print_list:name = h1
  info:libplutils:1031:readconf.c:388:configlist_print_list:name = h2
8 ...
9 info:libplutils:1031:readconf.c:388:configlist_print_list:name = h3
  info:libplutils:1031:readconf.c:388:configlist_print_list:name = joshua
10 ...
11 info:libplutils:1031:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
12 ...
13 ...
  info:libplutils:1031:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
14 ...
15 info:libplutils:1031:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
  info:libplutils:1031:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
16 ...
17 ...
  info:libplutils:1031:readconf.c:388:configlist_print_list:
18     name = /home/ronald/parallelplugins/data/montecarlo.input
  info:libplutils:1031:readconf.c:388:configlist_print_list:
20     name = /home/ronald/parallelplugins/data/image.input
21 ...
22 ...
23 info:libppm:1031:ppm.c:570:ppm_start:mode = 1
  info:libppm:1031:ppm.c:572:ppm_start:ppmnode = kid
24 ...
25 info:libppm:1031:ppm.c:716:ppm_load_harnesskernel_replicated:
26     start loading harness kernel and parallel plug-in
27
```

A. Appendix

```
info:libppm:1031:ppm.c:1422:ppm_read_integralinput:start reading input file
29 ...
info:libppm:1031:ppm.c:1459:ppm_read_integralinput:input file is opened
31 info:libppm:1031:ppm.c:1580:ppm_schedule_integral:start scheduling
...
33 info:libintegral:4813:integral.c:202:integral_init:libintegral is starting
info:libintegral:4813:integral.c:416:integral_init_rmix:start initialization of RMIX
35 ...
info:libintegral:4855:integral.c:202:integral_init:libintegral is starting
37 info:libintegral:4855:integral.c:416:integral_init_rmix:start initialization of RMIX
...
39 info:libintegral:4855:integral.c:451:integral_init_rmix:
    RMIX initialized and integral plug-in exported
41 ...
info:libintegral:4813:integral.c:451:integral_init_rmix:
43     RMIX initialized and integral plug-in exported
...
45 info:libintegral:3031:integral.c:202:integral_init:libintegral is starting
info:libintegral:3031:integral.c:416:integral_init_rmix:start initialization of RMIX
47 ...
info:libintegral:4785:integral.c:202:integral_init:libintegral is starting
49 info:libintegral:4785:integral.c:416:integral_init_rmix:start initialization of RMIX
...
51 info:libintegral:4785:integral.c:451:integral_init_rmix:
    RMIX initialized and integral plug-in exported
53 ...
info:libintegral:3031:integral.c:451:integral_init_rmix:
55     RMIX initialized and integral plug-in exported
...
57 info:libintegral:4855:integral.c:659:rmixintegral_integration:integral is -1.667648
info:libintegral:4813:integral.c:659:rmixintegral_integration:integral is -3.665542
59 info:libintegral:4785:integral.c:659:rmixintegral_integration:integral is -3.669111
info:libintegral:3031:integral.c:659:rmixintegral_integration:integral is -1.663693
61 ...
info:libppm:1031:ppm.c:2126:ppm_schedule_integral:+++++
63 info:libppm:1031:ppm.c:2128:ppm_schedule_integral:integral = -10.665995
info:libppm:1031:ppm.c:2130:ppm_schedule_integral:+++++
65 ...
info:libintegral:4855:integral.c:321:integral_fini:libintegral is shutting down
67 info:libintegral:4855:integral.c:469:integral_fini_rmix:start finalization of RMIX
...
69 info:libintegral:4813:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:4813:integral.c:469:integral_fini_rmix:start finalization of RMIX
71 ...
info:libintegral:4785:integral.c:321:integral_fini:libintegral is shutting down
73 info:libintegral:4785:integral.c:469:integral_fini_rmix:start finalization of RMIX
...
75 info:libintegral:3031:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:3031:integral.c:469:integral_fini_rmix:start finalization of RMIX
77 ...
info:libintegral:4855:integral.c:499:integral_fini_rmix:
79     RMIX finalized and integral plug-in unexported
...
81 info:libintegral:4813:integral.c:499:integral_fini_rmix:
    RMIX finalized and integral plug-in unexported
83 ...
info:libintegral:4785:integral.c:499:integral_fini_rmix:
85     RMIX finalized and integral plug-in unexported
...
87 info:libintegral:3031:integral.c:499:integral_fini_rmix:
    RMIX finalized and integral plug-in unexported
```

A. Appendix

A.2.1.2. Performed on Three Nodes and a Unavailable Node

```
ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
2 ...
info:libppm:28193:ppm.c:231:ppm_init:libppm is starting
4 info:libppm:28193:ppm.c:436:ppm_init_rmix:start initialization of RMIX
...
6 info:libppm:28193:ppm.c:471:ppm_init_rmix:RMIX initialized
info:libplutils:28193:readconf.c:114:configuration_read_file:start reading configuration file
8 info:libplutils:28193:readconf.c:388:configlist_print_list:name = h1
info:libplutils:28193:readconf.c:388:configlist_print_list:name = martha
10 info:libplutils:28193:readconf.c:388:configlist_print_list:name = h2
info:libplutils:28193:readconf.c:388:configlist_print_list:name = h3
12 ...
info:libplutils:28193:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
14 ...
info:libplutils:28193:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
16 info:libplutils:28193:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
info:libplutils:28193:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
18 ...
info:libplutils:28193:readconf.c:388:configlist_print_list:
20     name = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:28193:readconf.c:388:configlist_print_list:
22     name = /home/ronald/parallelplugins/data/image.input
...
24 info:libppm:28193:ppm.c:570:ppm_start:mode = 1
info:libppm:28193:ppm.c:572:ppm_start:ppmnode = kid
26 info:libppm:28193:ppm.c:716:ppm_load_harnesskernel_replicated:
    start loading harness kernel and parallel plug-in
28 info:libppm:28193:ppm.c:1422:ppm_read_integralinput:start reading input file
...
30 info:libppm:28193:ppm.c:1459:ppm_read_integralinput:input file is opened
ssh: martha: Name or service not known
32 ...
info:libppm:28193:ppm.c:1580:ppm_schedule_integral:start scheduling
34 ...
info:libintegral:6388:integral.c:202:integral_init:libintegral is starting
36 ...
info:libintegral:6388:integral.c:416:integral_init_rmix:start initialization of RMIX
38 info:libintegral:6534:integral.c:202:integral_init:libintegral is starting
...
40 info:libintegral:6534:integral.c:416:integral_init_rmix:start initialization of RMIX
...
42 info:libintegral:6534:integral.c:451:integral_init_rmix:
    RMIX initialized and integral plug-in exported
44 ...
info:libintegral:6388:integral.c:451:integral_init_rmix:
46     RMIX initialized and integral plug-in exported
...
48 info:libintegral:4918:integral.c:202:integral_init:libintegral is starting
...
50 info:libintegral:4918:integral.c:416:integral_init_rmix:start initialization of RMIX
...
52 info:libintegral:4918:integral.c:451:integral_init_rmix:
    RMIX initialized and integral plug-in exported
54 ...
warn:librmix:28193:tcpip4.c:264:rmix_tcpip4_resolve:
56     unable to lookup local name 'martha'
warn:librmix-rpcx:28193:rmix-rpcx.c:2655:rmix_rpcx_remoteref_create:
58     unable to resolve address
warn:librmix:28193:remoteref.c:975:rmix_remoteref_create5:
60     unable to create remote object reference
info:librmix-rpcx:28193:rmix-rpcx.c:328:rmix_rpcx_fini:rmix-rpcx shutdown
```


A. Appendix

```
62 warn:librmix:28193:remoteref.c:1266:rmix_remoteref_create6:
    unable to create remote object reference
64 warn:libppm:28193:ppm.c:1671:ppm_schedule_integral:
    could not create remote object references
66 ...
info:libintegral:6388:integral.c:659:rmixintegral_integration:integral is -2.773682
68 info:libintegral:6534:integral.c:659:rmixintegral_integration:integral is -5.132638
info:libintegral:4918:integral.c:659:rmixintegral_integration:integral is -2.756107
70 ...
info:libppm:28193:ppm.c:2126:ppm_schedule_integral:+++++
72 info:libppm:28193:ppm.c:2128:ppm_schedule_integral:integral = -10.662427
info:libppm:28193:ppm.c:2130:ppm_schedule_integral:+++++
74 ...
warn:libppm:28193:ppm.c:1236:ppm_createremoteref:
76     could not create remote object references
warn:libppm:28193:ppm.c:1142:ppm_unload_harnesskernel_soft:
78     could not create remote object references for martha
...
80 info:libintegral:6388:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:6388:integral.c:469:integral_fini_rmix:start finalization of RMIX
82 ...
info:libintegral:6534:integral.c:321:integral_fini:libintegral is shutting down
84 info:libintegral:6534:integral.c:469:integral_fini_rmix:start finalization of RMIX
...
86 info:libintegral:4918:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:4918:integral.c:469:integral_fini_rmix:start finalization of RMIX
88 ...
info:libintegral:6388:integral.c:499:integral_fini_rmix:
90     RMIX finalized and integral plug-in unexported
...
92 info:libintegral:6534:integral.c:499:integral_fini_rmix:
    RMIX finalized and integral plug-in unexported
94 ...
info:libintegral:4918:integral.c:499:integral_fini_rmix:
96     RMIX finalized and integral plug-in unexported
```

A.2.1.3. Performed on Three Nodes and the Second Node Fails

```
ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
2 ...
info:libppm:4936:ppm.c:200:ppm_init:libppm is starting
4 info:libppm:4936:ppm.c:391:ppm_init_rmix:start initialization of RMIX
...
6 info:libppm:4936:ppm.c:404:ppm_init_rmix:RMIX initialized
info:libplutils:4936:readconf.c:110:configuration_read_file:start reading configuration file
8 info:libplutils:4936:readconf.c:372:configlist_print_list:name = h1
info:libplutils:4936:readconf.c:372:configlist_print_list:name = h2
10 info:libplutils:4936:readconf.c:372:configlist_print_list:name = h3
...
12 info:libplutils:4936:readconf.c:372:configlist_print_list:name = libintegral.0.0.0.so
...
14 info:libplutils:4936:readconf.c:372:configlist_print_list:name = libimgproc.0.0.0.so
info:libplutils:4936:readconf.c:372:configlist_print_list:name = libimgproc.0.0.0.so
16 info:libplutils:4936:readconf.c:372:configlist_print_list:name = libimgproc.0.0.0.so
...
18 info:libplutils:4936:readconf.c:372:configlist_print_list:
    name = /home/ronald/parallelplugins/data/montecarlo.input
20 info:libplutils:4936:readconf.c:372:configlist_print_list:
    name = /home/ronald/parallelplugins/data/image.input
22 ...
info:libppm:4936:ppm.c:482:ppm_start:mode = 1
24 info:libppm:4936:ppm.c:614:ppm_load_harnesskernel_replicated:
```

A. Appendix

```
start loading harness kernel and parallel plug-in
26 info:libppm:4936:ppm.c:1120:ppm_read_integralinput:start reading input file
...
28 info:libppm:4936:ppm.c:1157:ppm_read_integralinput:input file is opened
info:libppm:4936:ppm.c:1278:ppm_schedule_integral:start scheduling
30 ...
info:libintegral:4892:integral.c:406:integral_init_rmix:start initialization of RMIX
32 ...
info:libintegral:4892:integral.c:441:integral_init_rmix:
34 RMIX initialized and integral plug-in exported
...
36 info:libintegral:17897:integral.c:202:integral_init:libintegral is starting
...
38 info:libintegral:17897:integral.c:406:integral_init_rmix:start initialization of RMIX
...
40 info:libintegral:18076:integral.c:202:integral_init:libintegral is starting
...
42 info:libintegral:18076:integral.c:406:integral_init_rmix:start initialization of RMIX
...
44 info:libintegral:18076:integral.c:441:integral_init_rmix:
RMIX initialized and integral plug-in exported
46 ...
info:libintegral:17897:integral.c:441:integral_init_rmix:
48 RMIX initialized and integral plug-in exported
...
50 warn:librmix-rpcx:4936:stream.c:1492:rmix_rpcx_stream_read_reply2:
unable to read procedure return
52 warn:librmix-rpcx:4936:rmix-rpcx.c:1662:rmix_rpcx_async_retrieve_job:
unable to read procedure return
54 ...
info:libintegral:18076:integral.c:649:rmixintegral_integration:integral is -2.772686
56 info:libintegral:17897:integral.c:649:rmixintegral_integration:integral is -2.769724
warn:librmix-rpcx:4936:rmix-rpcx.c:1909:rmix_rpcx_retrieve:unable to retrieve call return
58 warn:librmix:4936:rmix.c:1445:rmix_retrieve:unable to invoke method at a remote object
...
60 warn:1:4936:ppm.c:1086:rmixintegralclient_integration_retrieve:
unable to invoke remote object method
62 ...
found error, errors = 1
64 ...
info:libppm:4936:ppm.c:1594:ppm_schedule_integral:starting error recovery
66 ...
errors = 1
68 ...
lower = -0.666667
70 ...
upper = 0.666667
72 ...
resent
74 ...
info:libintegral:18076:integral.c:649:rmixintegral_integration:integral is -5.134623
76 ...
info:libppm:4936:ppm.c:1824:ppm_schedule_integral:+++++
78 info:libppm:4936:ppm.c:1826:ppm_schedule_integral:integral = -10.677033
info:libppm:4936:ppm.c:1828:ppm_schedule_integral:+++++
80 ...
info:libintegral:18076:integral.c:311:integral_fini:libintegral is shutting down
82 info:libintegral:18076:integral.c:459:integral_fini_rmix:start finalization of RMIX
...
84 info:libintegral:17897:integral.c:311:integral_fini:libintegral is shutting down
info:libintegral:17897:integral.c:459:integral_fini_rmix:start finalization of RMIX
86 ...
info:libintegral:18076:integral.c:489:integral_fini_rmix:
```

A. Appendix

```
88     RMIX finalized and integral plug-in unexported
...
90 info: libintegral:17897:integral.c:489:integral_fini_rmix:
    RMIX finalized and integral plug-in unexported
```

A.2.2. Output Listings of the Parallel Plug-in for Image Processing

A.2.2.1. Performed on Three Nodes

```
1 ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
3 info: libppm:11632:ppm.c:231:ppm_init:libppm is starting
  info: libppm:11632:ppm.c:436:ppm_init_rmix: start initialization of RMIX
5 ...
  info: libppm:11632:ppm.c:471:ppm_init_rmix:RMIX initialized
7 info: libplutils:11632:readconf.c:114:configuration_read_file: start reading configuration file
  info: libplutils:11632:readconf.c:388: configlist_print_list:name = h1
9 info: libplutils:11632:readconf.c:388: configlist_print_list:name = h2
  info: libplutils:11632:readconf.c:388: configlist_print_list:name = h3
11 info: libplutils:11632:readconf.c:388: configlist_print_list:name = joshua
...
13 info: libplutils:11632:readconf.c:388: configlist_print_list:name = libintegral.0.0.0.so
...
15 info: libplutils:11632:readconf.c:388: configlist_print_list:name = libimgproc.0.0.0.so
  info: libplutils:11632:readconf.c:388: configlist_print_list:name = libimgproc.0.0.0.so
17 info: libplutils:11632:readconf.c:388: configlist_print_list:name = libimgproc.0.0.0.so
...
19 info: libplutils:11632:readconf.c:388: configlist_print_list:
    name = /home/ronald/parallelplugins/data/montecarlo.input
21 info: libplutils:11632:readconf.c:388: configlist_print_list:
    name = /home/ronald/parallelplugins/data/image.input
23 ...
  info: libppm:11632:ppm.c:570:ppm_start:mode = 2
25 info: libppm:11632:ppm.c:572:ppm_start:ppmnode = kid
  info: libppm:11632:ppm.c:807:ppm_load_harnesskernel_distributed:
27     start loading harness kernel and distributed plug-in
...
29 info: libimgproc:30293:imgproc.c:275:imgproc_init:libimgproc is starting
  info: libimgproc:30293:imgproc.c:467:imgproc_init_rmix: start initialization of RMIX
31 ...
  info: libimgproc:30293:imgproc.c:500:imgproc_init_rmix:
33     RMIX initialized and imgproc plug-in exported
...
35 info: libimgproc:21029:imgproc.c:275:imgproc_init:libimgproc is starting
  info: libimgproc:21029:imgproc.c:467:imgproc_init_rmix: start initialization of RMIX
37 ...
  info: libimgproc:21029:imgproc.c:500:imgproc_init_rmix:
39     RMIX initialized and imgproc plug-in exported
...
41 info: libimgproc:4666:imgproc.c:275:imgproc_init:libimgproc is starting
  info: libimgproc:4666:imgproc.c:467:imgproc_init_rmix: start initialization of RMIX
43 ...
  info: libimgproc:4666:imgproc.c:500:imgproc_init_rmix:
45     RMIX initialized and imgproc plug-in exported
...
47 info: libppm:11632:ppm.c:2459:ppm_read_imageinput: start reading imgproc input file
...
49 info: libppm:11632:ppm.c:2200:ppm_imageprocessing_loader:
```

A. Appendix

```
source dir /home/ronald/parallelplugins/images
51 info:libppm:11632:ppm.c:2202:ppm_imageprocessing_loader:
    target dir /home/ronald/parallelplugins/imagetarget
53 ...
info:libimgproc:30293:imgproc.c:2564:imgproc_get_files:3 images found
55 ...
info:libimgproc:30293:imgproc.c:2654:imgproc_loadimages:reading file ngc2237_1024.jpg
57 info:libimgproc:30293:imgproc.c:2694:imgproc_loadimages:
    full filename is /home/ronald/parallelplugins/images/ngc2237_1024.jpg
59 ...
info:libimgproc:30293:imgproc.c:2864:imgproc_create_magickwand:
61 image ngc2237_1024.jpg successfully sent to h2
info:libimgproc:30293:imgproc.c:2654:imgproc_loadimages:reading file image.gif
63 info:libimgproc:30293:imgproc.c:2694:imgproc_loadimages:
    full filename is /home/ronald/parallelplugins/images/image.gif
65 ...
info:libimgproc:30293:imgproc.c:2864:imgproc_create_magickwand:
67 image image.gif successfully sent to h2
info:libimgproc:30293:imgproc.c:2654:imgproc_loadimages:reading file mapimg.gif
69 info:libimgproc:30293:imgproc.c:2694:imgproc_loadimages:
    full filename is /home/ronald/parallelplugins/images/mapimg.gif
71 ...
info:libimgproc:30293:imgproc.c:2864:imgproc_create_magickwand:
73 image mapimg.gif successfully sent to h2
...
75 info:libimgproc:21029:imgproc.c:1434:rmiximgproc_passimage:
    image ngc2237_1024.jpg successfully sent to h3
77 ...
info:libimgproc:4666:imgproc.c:1526:rmiximgproc_passimage:
79 acknowledgement for ngc2237_1024.jpg successfully sent to h2
...
81 info:libimgproc:21029:imgproc.c:1743:rmiximgproc_imageprocessed:
    acknowledgement for ngc2237_1024.jpg successfully sent to h1
83 ...
info:libimgproc:21029:imgproc.c:1434:rmiximgproc_passimage:
85 image image.gif successfully sent to h3
...
87 info:libimgproc:21029:imgproc.c:1434:rmiximgproc_passimage:
    image mapimg.gif successfully sent to h3
89 ...
info:libimgproc:4666:imgproc.c:1526:rmiximgproc_passimage:
91 acknowledgement for mapimg.gif successfully sent to h2
...
93 info:libimgproc:21029:imgproc.c:1743:rmiximgproc_imageprocessed:
    acknowledgement for mapimg.gif successfully sent to h1
95 ...
info:libimgproc:4666:imgproc.c:1526:rmiximgproc_passimage:
97 acknowledgement for image.gif successfully sent to h2
...
99 info:libimgproc:21029:imgproc.c:1743:rmiximgproc_imageprocessed:
    acknowledgement for image.gif successfully sent to h1
101 ...
info:libimgproc:4666:imgproc.c:371:imgproc_fini:libimgproc is shutting down
103 info:libimgproc:4666:imgproc.c:519:imgproc_fini_rmix:start finalization of RMIX
...
105 info:libimgproc:30293:imgproc.c:371:imgproc_fini:libimgproc is shutting down
info:libimgproc:30293:imgproc.c:519:imgproc_fini_rmix:start finalization of RMIX
107 info:libimgproc:21029:imgproc.c:371:imgproc_fini:libimgproc is shutting down
info:libimgproc:21029:imgproc.c:519:imgproc_fini_rmix:start finalization of RMIX
109 ...
info:libimgproc:4666:imgproc.c:549:imgproc_fini_rmix:
111 RMIX finalized and imgproc plug-in unexported
...
```

A. Appendix

```
113 info:libimgproc:21029:imgproc.c:549:imgproc_fini_rmix:
      RMX finalized and imgproc plug-in unexported
115 ...
info:libimgproc:30293:imgproc.c:549:imgproc_fini_rmix:
117      RMX finalized and imgproc plug-in unexported
```

A.2.2.2. Restoration of a Broken Image Processing Pipeline (1)

```
1 ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
3 info:libppm:14502:ppm.c:231:ppm_init:libppm is starting
info:libppm:14502:ppm.c:436:ppm_init_rmix:start initialization of RMX
5 ...
info:libppm:14502:ppm.c:471:ppm_init_rmix:RMX initialized
7 info:libplutils:14502:readconf.c:114:configuration_read_file:
  start reading configuration file
9 info:libplutils:14502:readconf.c:388:configlist_print_list:name = h1
info:libplutils:14502:readconf.c:388:configlist_print_list:name = h2
11 info:libplutils:14502:readconf.c:388:configlist_print_list:name = h3
info:libplutils:14502:readconf.c:388:configlist_print_list:name = joshua
13 ...
info:libplutils:14502:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
15 ...
info:libplutils:14502:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
17 info:libplutils:14502:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
info:libplutils:14502:readconf.c:388:configlist_print_list:name = libimgproc.0.0.0.so
19 ...
info:libplutils:14502:readconf.c:388:configlist_print_list:
21   name = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:14502:readconf.c:388:configlist_print_list:
23   name = /home/ronald/parallelplugins/data/image.input
...
25 info:libppm:14502:ppm.c:570:ppm_start:mode = 2
info:libppm:14502:ppm.c:572:ppm_start:ppmnode = kid
27 info:libppm:14502:ppm.c:807:ppm_load_harnesskernel_distributed:
  start loading harness kernel and distributed plug-in
29 ...
info:libimgproc:12137:imgproc.c:275:imgproc_init:libimgproc is starting
31 info:libimgproc:12137:imgproc.c:478:imgproc_init_rmix:start initialization of RMX
...
33 info:libimgproc:12137:imgproc.c:511:imgproc_init_rmix:
      RMX initialized and imgproc plug-in exported
35 ...
info:libimgproc:4331:imgproc.c:275:imgproc_init:libimgproc is starting
37 info:libimgproc:4331:imgproc.c:478:imgproc_init_rmix:start initialization of RMX
...
39 info:libimgproc:4331:imgproc.c:511:imgproc_init_rmix:
      RMX initialized and imgproc plug-in exported
41 ...
info:libimgproc:23875:imgproc.c:275:imgproc_init:libimgproc is starting
43 info:libimgproc:23875:imgproc.c:478:imgproc_init_rmix:start initialization of RMX
...
45 info:libimgproc:23875:imgproc.c:511:imgproc_init_rmix:
      RMX initialized and imgproc plug-in exported
47 ...
info:libppm:14502:ppm.c:2459:ppm_read_imageinput:start reading imgproc input file
49 info:libppm:14502:ppm.c:2200:ppm_imageprocessing_loader:
  source dir /home/ronald/parallelplugins/images
51 info:libppm:14502:ppm.c:2202:ppm_imageprocessing_loader:
  target dir /home/ronald/parallelplugins/imagetarget
53 ...
info:libimgproc:12137:imgproc.c:2664:imgproc_get_files:3 images found
```

A. Appendix

```
55 ...
info: libimgproc:12137:imgproc.c:2756:imgproc_loadimages:reading file ngc2237_1024.jpg
57 info: libimgproc:12137:imgproc.c:2796:imgproc_loadimages:
    full filename is /home/ronald/parallelplugins/images/ngc2237_1024.jpg
59 ...
info: libimgproc:12137:imgproc.c:2934:imgproc_create_magickwand:
61     image ngc2237_1024.jpg successfully sent to h2
...
63 info: libppm:14502:ppm.c:2891:rmixppm_repairpipe:*****
info: libppm:14502:ppm.c:2893:rmixppm_repairpipe:repair pipe called missing node = h2
65 info: libppm:14502:ppm.c:2901:rmixppm_repairpipe:missing node h2 found in list at position 1
info: libppm:14502:ppm.c:2908:rmixppm_repairpipe:1 free nodes available
67 info: libppm:14502:ppm.c:2933:rmixppm_repairpipe:
    name of the plug-in to reload is libimgproc.0.0.0.so
69 info: libppm:14502:ppm.c:807:ppm_load_harnesskernel_distributed:
    start loading harness kernel and distributed plug-in
71 ...
info: libimgproc:864:imgproc.c:275:imgproc_init:libimgproc is starting
73 info: libimgproc:864:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
75 info: libimgproc:864:imgproc.c:511:imgproc_init_rmix:
    RMIX initialized and imgproc plug-in exported
77 ...
info: libppm:14502:ppm.c:3286:rmixppm_repairpipe:*****
79 ...
info: libimgproc:12137:imgproc.c:2756:imgproc_loadimages:reading file image.gif
81 info: libimgproc:12137:imgproc.c:2796:imgproc_loadimages:
    full filename is /home/ronald/parallelplugins/images/image.gif
83 info: libimgproc:12137:imgproc.c:3255:imgproc_worklist_printlist:name = ngc2237_1024.jpg
...
85 info: libimgproc:12137:imgproc.c:2231:rmiximgproc_sendworklist: resend ngc2237_1024.jpg
...
87 info: libimgproc:12137:imgproc.c:2934:imgproc_create_magickwand:
    image image.gif successfully sent to joshua
89 info: libimgproc:12137:imgproc.c:2756:imgproc_loadimages:reading file mapimg.gif
info: libimgproc:12137:imgproc.c:2796:imgproc_loadimages:
91     full filename is /home/ronald/parallelplugins/images/mapimg.gif
...
93 info: libimgproc:12137:imgproc.c:2934:imgproc_create_magickwand:
    image mapimg.gif successfully sent to joshua
95 ...
info: libimgproc:4331:imgproc.c:1445:rmiximgproc_passimage:
97     image ngc2237_1024.jpg successfully sent to h3
...
99 info: libimgproc:864:imgproc.c:1445:rmiximgproc_passimage:
    image ngc2237_1024.jpg successfully sent to h3
101 info: libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
    image ngc2237_1024.jpg successfully stored
103 ...
info: libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
105     acknowledgement ngc2237_1024.jpg successfully sent to joshua
...
107 info: libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
    acknowledgement for ngc2237_1024.jpg successfully sent to h1
109 ...
info: libimgproc:864:imgproc.c:1445:rmiximgproc_passimage:
111     image image.gif successfully sent to h3
info: libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
113     image ngc2237_1024.jpg successfully stored
...
115 info: libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
    acknowledgement ngc2237_1024.jpg successfully sent to joshua
117 ...
```

A. Appendix

```
info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
119   acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
121 info:libimgproc:864:imgproc.c:1445:rmiximgproc_passimage:
    image mapimg.gif successfully sent to h3
123 info:libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
    image mapimg.gif successfully stored
125 ...
info:libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
127   acknowledgement mapimg.gif successfully sent to joshua
...
129 info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
    acknowledgement for mapimg.gif successfully sent to h1
131 ...
info:libimgproc:23875:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
133 info:libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
    image image.gif successfully stored
135 ...
info:libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
137   acknowledgement image.gif successfully sent to joshua
...
139 info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
    acknowledgement for image.gif successfully sent to h1
141 ...
info:libimgproc:864:imgproc.c:382:imgproc_fini:libimgproc is shutting down
143 info:libimgproc:864:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
...
145 info:libimgproc:12137:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info:libimgproc:12137:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
147 ...
info:libimgproc:864:imgproc.c:560:imgproc_fini_rmix:
149   RMIX finalized and imgproc plug-in unexported
info:libimgproc:12137:imgproc.c:560:imgproc_fini_rmix:
151   RMIX finalized and imgproc plug-in unexported
...
153 info:libimgproc:4331:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info:libimgproc:4331:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
155 ...
info:libimgproc:4331:imgproc.c:560:imgproc_fini_rmix:
157   RMIX finalized and imgproc plug-in unexported
...
159 info:libppm:14502:ppm.c:335:ppm_fini:libppm is shutting down
info:libppm:14502:ppm.c:489:ppm_fini_rmix:start finalization of RMIX
161 ...
info:libppm:14502:ppm.c:519:ppm_fini_rmix:RMIX finalized
163 ...
```

A.2.2.3. Restoration of a Broken Image Processing Pipeline (2)

```
1 ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
3 info:libppm:14868:ppm.c:231:ppm_init:libppm is starting
info:libppm:14868:ppm.c:436:ppm_init_rmix:start initialization of RMIX
5 ...
info:libppm:14868:ppm.c:471:ppm_init_rmix:RMIX initialized
7 info:libplutils:14868:readconf.c:114:configuration_read_file:start reading configuration file
info:libplutils:14868:readconf.c:388:configlist_print_list:name = h1
9 info:libplutils:14868:readconf.c:388:configlist_print_list:name = h2
info:libplutils:14868:readconf.c:388:configlist_print_list:name = h3
11 info:libplutils:14868:readconf.c:388:configlist_print_list:name = joshua
...
13 info:libplutils:14868:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
```

A. Appendix

```
...
15 info: libplutils:14868:readconf.c:388: configlist_print_list: name = libimgproc.0.0.0.so
info: libplutils:14868:readconf.c:388: configlist_print_list: name = libimgproc.0.0.0.so
17 info: libplutils:14868:readconf.c:388: configlist_print_list: name = libimgproc.0.0.0.so
...
19 info: libplutils:14868:readconf.c:388: configlist_print_list:
    name = /home/ronald/parallelplugins/data/montecarlo.input
21 info: libplutils:14868:readconf.c:388: configlist_print_list:
    name = /home/ronald/parallelplugins/data/image.input
23 ...
info: libppm:14868:ppm.c:570: ppm_start: mode = 2
25 info: libppm:14868:ppm.c:572: ppm_start: ppmnode = kid
info: libppm:14868:ppm.c:807: ppm_load_harnesskernel_distributed:
27     start loading harness kernel and distributed plug-in
...
29 info: libimgproc:27276:imgproc.c:275: imgproc_init: libimgproc is starting
info: libimgproc:27276:imgproc.c:478: imgproc_init_rmix: start initialization of RMIX
31 ...
info: libimgproc:27276:imgproc.c:511: imgproc_init_rmix:
33     RMIX initialized and imgproc plug-in exported
...
35 info: libimgproc:20158:imgproc.c:275: imgproc_init: libimgproc is starting
info: libimgproc:20158:imgproc.c:478: imgproc_init_rmix: start initialization of RMIX
37 ...
info: libimgproc:20158:imgproc.c:511: imgproc_init_rmix:
39     RMIX initialized and imgproc plug-in exported
...
41 info: libimgproc:31774:imgproc.c:275: imgproc_init: libimgproc is starting
info: libimgproc:31774:imgproc.c:478: imgproc_init_rmix: start initialization of RMIX
43 ...
info: libimgproc:31774:imgproc.c:511: imgproc_init_rmix:
45     RMIX initialized and imgproc plug-in exported
...
47 info: libppm:14868:ppm.c:2459: ppm_read_imageinput: start reading imgproc input file
info: libppm:14868:ppm.c:2474: ppm_read_imageinput:
49     more than one input file entries found - try using second one
info: libppm:14868:ppm.c:2200: ppm_imageprocessing_loader:
51     source dir /home/ronald/parallelplugins/images
info: libppm:14868:ppm.c:2202: ppm_imageprocessing_loader:
53     target dir /home/ronald/parallelplugins/imagetarget
...
55 info: libimgproc:27276:imgproc.c:2899: imgproc_get_files: 3 images found
...
57 info: libimgproc:27276:imgproc.c:2991: imgproc_loadimages: reading file ngc2237_1024.jpg
info: libimgproc:27276:imgproc.c:3031: imgproc_loadimages:
59     full filename is /home/ronald/parallelplugins/images/ngc2237_1024.jpg
...
61 info: libimgproc:27276:imgproc.c:3237: imgproc_create_magickwand:
    image ngc2237_1024.jpg successfully sent to h2
63 info: libimgproc:27276:imgproc.c:2991: imgproc_loadimages: reading file image.gif
info: libimgproc:27276:imgproc.c:3031: imgproc_loadimages:
65     full filename is /home/ronald/parallelplugins/images/image.gif
...
67 info: libimgproc:27276:imgproc.c:3237: imgproc_create_magickwand:
    image image.gif successfully sent to h2
69 info: libimgproc:27276:imgproc.c:2991: imgproc_loadimages: reading file mapimg.gif
info: libimgproc:27276:imgproc.c:3031: imgproc_loadimages:
71     full filename is /home/ronald/parallelplugins/images/mapimg.gif
...
73 info: libimgproc:27276:imgproc.c:3237: imgproc_create_magickwand:
    image mapimg.gif successfully sent to h2
75 ...
info: libimgproc:20158:imgproc.c:1510: rmiximgproc_passimage:
```


A. Appendix

```
77     image ngc2237_1024.jpg successfully sent to h3
info: libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
79     image ngc2237_1024.jpg successfully stored
warn: librmix-rpcx:27276:stream.c:1492:rmix_rpcx_stream_read_reply2:
81     unable to read procedure return
warn: librmix-rpcx:27276:rmix-rpcx.c:1207:rmix_rpcx_oneway_retrieve_job:
83     unable to read procedure return
warn: librmix-rpcx:27276:stream.c:1492:rmix_rpcx_stream_read_reply2:
85     unable to read procedure return
warn: librmix-rpcx:27276:rmix-rpcx.c:1207:rmix_rpcx_oneway_retrieve_job:
87     unable to read procedure return
...
89 warn: librmix:31774:tcpip4.c:1074:rmix_tcpip4_client_open:unable to connect client socket
warn: librmix:31774:tcpip4.c:1149:rmix_tcpip4_client_open2:unable to open client
91 warn: librmix:31774:tcpip4.c:1209:rmix_tcpip4_client_open3:unable to open client
warn: librmix-rpcx:31774:rmix-rpcx.c:864:rmix_rpcx_oneway:unable to open TCP/IPv4 client
93 warn: librmix:31774:rmix.c:1134:rmix_oneway:unable to invoke method at a remote object
...
95 warn: 1:31774:imgproc.c:2058:rmiximgprocclient_imageprocessed_oneway:
    unable to invoke remote object method
97 warn: libimgproc:31774:imgproc.c:1625:rmiximgproc_passimage:
    could not call remote object for passing image
99 ...
info: libppm:14868:ppm.c:2891:rmixppm_repairpipe:*****
101 info: libppm:14868:ppm.c:2893:rmixppm_repairpipe:repair pipe called missing node = h2
info: libppm:14868:ppm.c:2901:rmixppm_repairpipe:missing node h2 found in list at position 1
103 info: libppm:14868:ppm.c:2908:rmixppm_repairpipe:1 free nodes available
info: libppm:14868:ppm.c:2933:rmixppm_repairpipe:
105     name of the plug-in to reload is libimgproc.0.0.0.so
info: libppm:14868:ppm.c:807:ppm_load_harnesskernel_distributed:
107     start loading harness kernel and distributed plug-in
...
109 info: libimgproc:17907:imgproc.c:275:imgproc_init:libimgproc is starting
info: libimgproc:17907:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
111 ...
info: libimgproc:17907:imgproc.c:511:imgproc_init_rmix:
113     RMIX initialized and imgproc plug-in exported
...
115 info: libppm:14868:ppm.c:3286:rmixppm_repairpipe:*****
info: libimgproc:27276:imgproc.c:3548:imgproc_worklist_printlist:name = ngc2237_1024.jpg
117 info: libimgproc:27276:imgproc.c:3548:imgproc_worklist_printlist:name = image.gif
info: libimgproc:27276:imgproc.c:3548:imgproc_worklist_printlist:name = mapimg.gif
119 ...
info: libimgproc:27276:imgproc.c:2466:rmiximgproc_sendworklistresend ngc2237_1024.jpg
121 ...
info: libimgproc:27276:imgproc.c:2466:rmiximgproc_sendworklist:resend image.gif
123 ...
info: libimgproc:27276:imgproc.c:2466:rmiximgproc_sendworklist:resend mapimg.gif
125 ...
info: libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
127     acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
129 info: libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
    acknowledgement ngc2237_1024.jpg successfully sent to joshua
131 ...
info: libimgproc:17907:imgproc.c:1510:rmiximgproc_passimage:
133     image ngc2237_1024.jpg successfully sent to h3
info: libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
135     image ngc2237_1024.jpg successfully stored
...
137 info: libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
    acknowledgement ngc2237_1024.jpg successfully sent to joshua
139 ...
```

A. Appendix

```
info: libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
141     acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
143 info: libimgproc:17907:imgproc.c:1510:rmiximgproc_passimage:
     image image.gif successfully sent to h3
145 ...
info: libimgproc:17907:imgproc.c:1510:rmiximgproc_passimage:
147     image mapimg.gif successfully sent to h3
info: libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
149     image mapimg.gif successfully stored
...
151 info: libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
     acknowledgement mapimg.gif successfully sent to joshua
153 ...
info: libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
155     acknowledgement for mapimg.gif successfully sent to h1
info: libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
157     image image.gif successfully stored
...
159 info: libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
     acknowledgement image.gif successfully sent to joshua
161 ...
info: libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
163     acknowledgement for image.gif successfully sent to h1
...
165 info: libimgproc:31774:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info: libimgproc:31774:imgproc.c:530:imgproc_fini_rmix: start finalization of RMX
167 ...
info: libimgproc:27276:imgproc.c:382:imgproc_fini:libimgproc is shutting down
169 info: libimgproc:27276:imgproc.c:530:imgproc_fini_rmix: start finalization of RMX
...
171 info: libimgproc:17907:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info: libimgproc:17907:imgproc.c:530:imgproc_fini_rmix: start finalization of RMX
173 ...
info: libimgproc:31774:imgproc.c:560:imgproc_fini_rmix:
175     RMX finalized and imgproc plug-in unexported
...
177 info: libimgproc:27276:imgproc.c:560:imgproc_fini_rmix:
     RMX finalized and imgproc plug-in unexported
179 ...
info: libimgproc:17907:imgproc.c:560:imgproc_fini_rmix:
181     RMX finalized and imgproc plug-in unexported
...
```

A.3. Source Listings

A.3.1. RMX Interface Descriptions

A.3.1.1. RMX Interface Description for the Parallel Plug-in Manager

```
/*
2 *
  * Header file for rmix parallel plug-in manager interface.
4 * Copyright (c) Ronald Baumann
  *
6 * For more information see the following files in the source distribution top-
  * level directory or package data directory (usually /usr/local/share/package):
```

A. Appendix

```
8 *
9 * - README    for general package information.
10 * - INSTALL   for package install information.
11 * - COPYING   for package license information and copying conditions.
12 * - AUTHORS   for package authors information.
13 * - ChangeLog for package changes information.
14 *
15 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
16 * top-level directory to create the target file.
17 *
18 *****/
19
20 /** \file rmixppm.h
21 * \brief Header file for defining the rmix signatures for the parallel plug-in
22 * manager.
23 *
24 * The header file contains the signatures for the communication functions of
25 * the parallel plug-in manger. The signatures define the parameters, which are
26 * sent between the stub functions on each side.
27 */
28
29
30 /* Avoid to include the content of this header file twice. */
31 #ifndef RMIXPPM_RMIXPPM_H
32 #define RMIXPPM_RMIXPPM_H
33
34
35 /******
36 *
37 * Macros
38 *
39 *****/
40
41 /* Flag for <harness-rmix.0/harness-rmix.h> header. */
42 #ifndef HARNESS_RMIX_HARNESS_RMIX_H
43 #ifndef HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
44 #define HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H 1
45 #endif
46 #endif
47
48
49
50 /******
51 *
52 * Method descriptor for repair pipeline method.
53 *
54 *****/
55
56 /** \def RMIXPPM_REPAIRPIPE_INPUT
57 * \brief Input signature for repair pipeline method.
58 */
59 #define RMIXPPM_REPAIRPIPE_INPUT\
60     RMIX_SIGNATURE(\
61         RMIX_SIGNATURE_TYPE(string,)) /* name of the node */
62
63
64 /** \def RMIXPPM_REPAIRPIPE_OUTPUT
65 * \brief Output signature for repair pipeline method.
66 */
67 #define RMIXPPM_REPAIRPIPE_OUTPUT\
68     RMIX_SIGNATURE(\
69         RMIX_SIGNATURE_TYPE(int,)) /* return */
70
```

A. Appendix

```
72 /** \def RMIXPPM_REPAIRPIPE_METHOD
73 * \brief Server-side descriptor for repair pipeline method.
74 */
75 #define RMIXPPM_REPAIRPIPE_METHOD\
76     RMIX_METHOD_INIT("repairpipe",\
77                     RMIXPPM_REPAIRPIPE_OUTPUT,\
78                     RMIXPPM_REPAIRPIPE_INPUT,\
79                     rmixppm_repairpipe_call)
80
81
82 /** \def RMIXPPM_REPAIRPIPE_METHOD_CLIENT
83 * \brief Client-side descriptor for repair pipeline method.
84 */
85 #define RMIXPPM_REPAIRPIPE_METHOD_CLIENT\
86     RMIX_METHOD_INIT("repairpipe",\
87                     RMIXPPM_REPAIRPIPE_OUTPUT,\
88                     RMIXPPM_REPAIRPIPE_INPUT,\
89                     NULL)
90
91
92 /*****
93 *
94 * Interface for rmixintegral.
95 *
96 *****/
97
98 /** \def RMIXPPM_METHODS
99 * \brief Server-side method descriptors.
100 */
101 #define RMIXPPM_METHODS \
102     { \
103         RMIXPPM_REPAIRPIPE_METHOD          /* repairpipe method */\
104     }
105
106
107 /** \def RMIXPPM_METHODS_CLIENT
108 * \brief Client-side method descriptors.
109 */
110 #define RMIXPPM_METHODS_CLIENT \
111     { \
112         RMIXPPM_REPAIRPIPE_METHOD_CLIENT  /* repairpipe method */\
113     }
114
115
116 /** \def RMIXPPM_METHODS_REPAIRPIPE_INDEX
117 * \brief Method indice repairpipe.
118 */
119 #define RMIXPPM_METHODS_REPAIRPIPE_INDEX 0 /* repairpipe method index */
120
121
122 /** \def RMIXPPM_METHODS_COUNT
123 * \brief Method indices for rmix repairpipe methods.
124 */
125 #define RMIXPPM_METHODS_COUNT 1 /* method count */
126
127
128 /*****
129 *
130 * Includes
131 *
132 *****/
133
134 /* Include <harness-rmix/harness-rmix.h> header. */
```

A. Appendix

```
134 #if HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
135 #include <harness-rmix/harness-rmix.h>
136 #endif
137
138
139 /*****
140 *
141 * Data
142 *
143 *****/
144
145 /** \var extern const rmix_method_t \
146     rmixppm_methods[RMIXPPM_METHODS_COUNT];
147     * \brief Server-side method descriptors for rmix ppm.
148 */
149 extern const rmix_method_t rmixppm_methods[RMIXPPM_METHODS_COUNT];
150
151
152 /** \var extern const rmix_interface_t rmixppm_interface;
153     * \brief Server-side interface for rmix ppm.
154 */
155 extern const rmix_interface_t rmixppm_interface;
156
157 #endif /* RMIXPPM_RMIXPPM_H */
158
159
160 /*****
161 *
162 * END OF FILE
163 *
164 *****/
```

A.3.1.2. RMIX Interface Description for the Monte Carlo Integration

```
1 /*****
2 *
3 * Header file for rmix integral interface.
4 * Copyright (c) Ronald Baumann
5 *
6 * For more information see the following files in the source distribution top-
7 * level directory or package data directory (usually /usr/local/share/package):
8 *
9 * - README for general package information.
10 * - INSTALL for package install information.
11 * - COPYING for package license information and copying conditions.
12 * - AUTHORS for package authors information.
13 * - ChangeLog for package changes information.
14 *
15 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
16 * top-level directory to create the target file.
17 *
18 *****/
19
20 /** \file rmixintegral.h
21 * \brief Header file for defining the rmix signatures for the integral
22 * communication functions.
23 *
24 * The header file contains the signatures for the communication functions of
25 * the Monte Carlo example. The signatures define the parameters, which are
26 * sent between the stub functions on each side.
27 */
```

A. Appendix

```
29 /* Avoid to include the content of this header file twice. */
31 #ifndef RMXINTEGRAL_RMXINTEGRAL_H
32 #define RMXINTEGRAL_RMXINTEGRAL_H
33
34 /*****
35 *
36 * Macros
37 *
38 *****/
39
40 /* Flag for <harness-rmix.0/harness-rmix.h> header. */
41 #ifndef HARNESS_RMX_HARNESS_RMX_H
42 #define HARNESS_RMX_HARNESS_RMX_H
43 #ifndef HAVE_HARNESS_RMX_0_HARNESS_RMX_H
44 #define HAVE_HARNESS_RMX_0_HARNESS_RMX_H 1
45 #endif
46 #endif
47
48 /*****
49 *
50 * Method descriptor for integration method.
51 *
52 *****/
53
54 /** \def RMXINTEGRAL_INTEGRATION_INPUT
55 * \brief Input signature for integration method.
56 */
57 #define RMXINTEGRAL_INTEGRATION_INPUT\
58     RMX_SIGNATURE(\
59         RMX_SIGNATURE_TYPE(unsigned_int, /* iterations */\
60         RMX_SIGNATURE_TYPE(double, /* lower boundary */\
61         RMX_SIGNATURE_TYPE(double, /* upper boundary */\
62         RMX_SIGNATURE_TYPE(variable_array(double), /* coefficients */\
63         ))))
64
65
66 /** \def RMXINTEGRAL_INTEGRATION_OUTPUT
67 * \brief Output signature for integration method.
68 */
69 #define RMXINTEGRAL_INTEGRATION_OUTPUT\
70     RMX_SIGNATURE(\
71         RMX_SIGNATURE_TYPE(int, /* return */\
72         RMX_SIGNATURE_TYPE(double, /* result */\
73
74
75
76 /** \def RMXINTEGRAL_INTEGRATION_METHOD
77 * \brief Server-side descriptor for integration method.
78 */
79 #define RMXINTEGRAL_INTEGRATION_METHOD\
80     RMX_METHOD_INIT("integration",\
81         RMXINTEGRAL_INTEGRATION_OUTPUT,\
82         RMXINTEGRAL_INTEGRATION_INPUT,\
83         rmixintegral_integration_call)
84
85
86 /** \def RMXINTEGRAL_INTEGRATION_METHOD_CLIENT
87 * \brief Client-side descriptor for integration method.
88 */
89 #define RMXINTEGRAL_INTEGRATION_METHOD_CLIENT\
90     RMX_METHOD_INIT("integration",\
```

A. Appendix

```
91         RMXINTEGRAL_INTEGRATION_OUTPUT,\
92         RMXINTEGRAL_INTEGRATION_INPUT,\
93         NULL)
94
95
96 /*****
97  *
98  * Interface for rmix integral.
99  *
100 *****/
101
102 /** \def RMXINTEGRAL_METHODS
103  * \brief Server-side method descriptors.
104  */
105 #define RMXINTEGRAL_METHODS \
106     { \
107         RMXINTEGRAL_INTEGRATION_METHOD /* integration method */\
108     }
109
110
111 /** \def RMXINTEGRAL_METHODS_CLIENT
112  * \brief Client-side method descriptors.
113  */
114 #define RMXINTEGRAL_METHODS_CLIENT \
115     { \
116         RMXINTEGRAL_INTEGRATION_METHOD_CLIENT /* integration method */\
117     }
118
119
120 /** \def RMXINTEGRAL_METHODS_INTEGRATION_INDEX
121  * \brief Method indice integration.
122  */
123 #define RMXINTEGRAL_METHODS_INTEGRATION_INDEX 0 /* integration method index */
124
125 /** \def RMXINTEGRAL_METHODS_COUNT
126  * \brief Method indices for rmix integration methods.
127  */
128 #define RMXINTEGRAL_METHODS_COUNT 1 /* method count */
129
130
131 /*****
132  *
133  * Includes
134  *
135 *****/
136
137 /* Include <harness-rmix/harness-rmix.h> header. */
138 #if HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
139 #include <harness-rmix/harness-rmix.h>
140 #endif
141
142
143 /*****
144  *
145  * Data
146  *
147 *****/
148
149 /** \var extern const rmix_method_t \
150         rmixintegral_methods[RMXINTEGRAL_METHODS_COUNT];
151  * \brief Server-side method descriptors for rmix integral.
152  */
153 extern const rmix_method_t rmixintegral_methods[RMXINTEGRAL_METHODS_COUNT];
```

A. Appendix

```
155 /** \var extern const rmix_interface_t rmixintegral_interface;
157 * \brief Server-side interface for rmix integral.
*/
159 extern const rmix_interface_t rmixintegral_interface;

161 #endif /* RMIXINTEGRAL_RMIXINTEGRAL_H */
163
165 /*
167 * END OF FILE
167 *
*****/
```

A.3.1.3. RMIX Interface Description for the Image Processing Pipeline

```
/*
2 *
3 * Header file for rmix image processing pipeline interface.
4 * Copyright (c) Ronald Baumann
5 *
6 * For more information see the following files in the source distribution top-
7 * level directory or package data directory (usually /usr/local/share/package):
8 *
9 * - README for general package information.
10 * - INSTALL for package install information.
11 * - COPYING for package license information and copying conditions.
12 * - AUTHORS for package authors information.
13 * - ChangeLog for package changes information.
14 *
15 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
16 * top-level directory to create the target file.
17 *
18 *****/

20 /** \file rmiximgproc.h
21 * \brief Header file for defining the rmix signatures for the image
22 * processing pipeline.
23 *
24 * The header file contains the signatures for the communication functions of
25 * the image processing example. The signatures define the parameters, which
26 * are sent between the stub functions on each side.
27 */
28

30 /* Avoid to include the content of this header file twice. */
31 #ifndef RMIXIMGPROC_RMIXIMGPROC_H
32 #define RMIXIMGPROC_RMIXIMGPROC_H

34
36 /*
38 * Macros
38 *
*****/

40 /* Flag for <harness-rmix.0/harness-rmix.h> header. */
41 #ifndef HARNESS_RMIX_HARNESS_RMIX_H
42 #ifndef HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
43 #define HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H 1
```


A. Appendix

```
#endif
46 #endif

48
49 /*****
50 *
51 * Method descriptor for initlisation of the pipeline method.
52 *
53 *****/
54
55 /** \def RMIXIMGPROC_INITPIPELINE_INPUT
56 * \brief Input signature for initpipeline method.
57 */
58 #define RMIXIMGPROC_INITPIPELINE_INPUT\
60     RMIX_SIGNATURE(\
61         RMIX_SIGNATURE_TYPE(unsigned_int,          /* filter selection */\
62         RMIX_SIGNATURE_TYPE(string,                /* source dir      */\
63         RMIX_SIGNATURE_TYPE(string,                /* traget dir     */\
64         RMIX_SIGNATURE_TYPE(string,                /* successor      */\
65         RMIX_SIGNATURE_TYPE(string,                /* predecessor    */\
66         )))
67
68
69 /** \def RMIXIMGPROC_INITPIPELINE_OUTPUT
70 * \brief Output signature for initpipeline method.
71 */
72 #define RMIXIMGPROC_INITPIPELINE_OUTPUT\
74     RMIX_SIGNATURE(\
75         RMIX_SIGNATURE_TYPE(int,))                /* return          */
76
77
78 /** \def RMIXIMGPROC_INITPIPELINE_METHOD
79 * \brief Server-side descriptor for initpipeline method.
80 */
81 #define RMIXIMGPROC_INITPIPELINE_METHOD\
82     RMIX_METHOD_INIT("initpipeline",\
83         RMIXIMGPROC_INITPIPELINE_OUTPUT,\
84         RMIXIMGPROC_INITPIPELINE_INPUT,\
85         rmiximgproc_initpipeline_call)
86
87
88 /** \def RMIXIMGPROC_INITPIPELINE_METHOD
89 * \brief Client-side descriptor for initpipeline method.
90 */
91 #define RMIXIMGPROC_INITPIPELINE_METHOD_CLIENT\
92     RMIX_METHOD_INIT("initpipeline",\
93         RMIXIMGPROC_INITPIPELINE_OUTPUT,\
94         RMIXIMGPROC_INITPIPELINE_INPUT,\
95         NULL)
96
97 /*****
98 *
99 * Method descriptor for invokepipeline method.
100 *
101 *****/
102
103 /** \def RMIXIMGPROC_INVOKEPIPELINE_INPUT
104 * \brief Input signature for invokepipeline method.
105 */
106 #define RMIXIMGPROC_INVOKEPIPELINE_INPUT\
107     RMIX_SIGNATURE()                                /* void           */
```

A. Appendix

```
108
110 /** \def RMIXIMGPROC_INVOKEPIPELINE_OUTPUT
    * \brief Output signature for invokepipeline method.
112 */
    #define RMIXIMGPROC_INVOKEPIPELINE_OUTPUT\
114     RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int ,))          /* return */
116
118 /** \def RMIXIMGPROC_INVOKEPIPELINE_METHOD
    * \brief Server-side descriptor for invokepipeline method.
120 */
    #define RMIXIMGPROC_INVOKEPIPELINE_METHOD\
122     RMIX_METHOD_INIT("invokepipeline",\
        RMIXIMGPROC_INVOKEPIPELINE_OUTPUT,\
124         RMIXIMGPROC_INVOKEPIPELINE_INPUT,\
            rmiximgproc_invokepipeline_call)
126
128 /** \def RMIXIMGPROC_INVOKEPIPELINE_METHOD_CLIENT
    * \brief Client-side descriptor for invokepipeline method.
130 */
    #define RMIXIMGPROC_INVOKEPIPELINE_METHOD_CLIENT\
132     RMIX_METHOD_INIT("invokepipeline",\
        RMIXIMGPROC_INVOKEPIPELINE_OUTPUT,\
134         RMIXIMGPROC_INVOKEPIPELINE_INPUT,\
            NULL)
136
138 /*****
    *
140 * Method descriptor for send backup list method.
    *
142 *****/
144 /** \def RMIXIMGPROC_SENDWORKLIST_INPUT
    * \brief Input signature for send backup list method.
146 */
    #define RMIXIMGPROC_SENDWORKLIST_INPUT\
148     RMIX_SIGNATURE()          /* void */
150
152 /** \def RMIXIMGPROC_SENDWORKLIST_OUTPUT
    * \brief Output signature for send backup list method.
    */
154 #define RMIXIMGPROC_SENDWORKLIST_OUTPUT\
    RMIX_SIGNATURE(\
156     RMIX_SIGNATURE_TYPE(int ,))          /* return */
158
160 /** \def RMIXIMGPROC_SENDWORKLIST_METHOD
    * \brief Server-side descriptor for send back up list method.
    */
162 #define RMIXIMGPROC_SENDWORKLIST_METHOD\
    RMIX_METHOD_INIT("sendworklist",\
164         RMIXIMGPROC_SENDWORKLIST_OUTPUT,\
            RMIXIMGPROC_SENDWORKLIST_INPUT,\
            rmiximgproc_sendworklist_call)
166
168 /** \def RMIXIMGPROC_SENDWORKLIST_METHOD_CLIENT
    * \brief Client-side descriptor for send backup list method.
170
```

A. Appendix

```
172 */
173 #define RMXIMGPROC_SENDWORKLIST_METHOD_CLIENT\
174     RMX_METHOD_INIT("sendworklist",\
175                     RMXIMGPROC_SENDWORKLIST_OUTPUT,\
176                     RMXIMGPROC_SENDWORKLIST_INPUT,\
177                     NULL)
178
179 /*****
180 *
181 * Method descriptor for availability check method.
182 *
183 *****/
184
185 /** \def RMXIMGPROC_AVAILABILITYCHECK_INPUT
186 * \brief Input signature for availability check method.
187 */
188 #define RMXIMGPROC_AVAILABILITYCHECK_INPUT\
189     RMX_SIGNATURE() /* void */
190
191 /** \def RMXIMGPROC_AVAILABILITYCHECK_OUTPUT
192 * \brief Output signature for availability check method.
193 */
194 #define RMXIMGPROC_AVAILABILITYCHECK_OUTPUT\
195     RMX_SIGNATURE(\
196         RMX_SIGNATURE_TYPE(int,)) /* return */
197
198
199 /** \def RMXIMGPROC_AVAILABILITYCHECK_METHOD
200 * \brief Server-side descriptor for availability check method.
201 */
202 #define RMXIMGPROC_AVAILABILITYCHECK_METHOD\
203     RMX_METHOD_INIT("availabilitycheck",\
204                     RMXIMGPROC_AVAILABILITYCHECK_OUTPUT,\
205                     RMXIMGPROC_AVAILABILITYCHECK_INPUT,\
206                     rmixmapproc_availabilitycheck_call)
207
208
209 /** \def RMXIMGPROC_AVAILABILITYCHECK_METHOD_CLIENT
210 * \brief Client-side descriptor for availability check method.
211 */
212 #define RMXIMGPROC_AVAILABILITYCHECK_METHOD_CLIENT\
213     RMX_METHOD_INIT("availabilitycheck",\
214                     RMXIMGPROC_AVAILABILITYCHECK_OUTPUT,\
215                     RMXIMGPROC_AVAILABILITYCHECK_INPUT,\
216                     NULL)
217
218
219 /*****
220 *
221 * Method descriptor for passimage method.
222 *
223 *****/
224
225 /** \def RMXIMGPROC_PASSIMAGE_INPUT
226 * \brief Input signature for pass image method.
227 */
228 #define RMXIMGPROC_PASSIMAGE_INPUT\
229     RMX_SIGNATURE(\
230         RMX_SIGNATURE_TYPE(string, /* image name */\
231         RMX_SIGNATURE_TYPE(variable_array(unsigned_char), /* image blob */\
232         )))
```

A. Appendix

```
234
236 /** \def RMIXIMGPROC_PASSIMAGE_OUTPUT
    * \brief Output signature for pass image method.
238 */
    #define RMIXIMGPROC_PASSIMAGE_OUTPUT\
240     RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))          /* return */
242
244 /** \def RMIXIMGPROC_PASSIMAGE_METHOD
    * \brief Server-side descriptor for pass image method.
246 */
    #define RMIXIMGPROC_PASSIMAGE_METHOD\
248     RMIX_METHOD_INIT("passimage",\
        RMIXIMGPROC_PASSIMAGE_OUTPUT,\
250         RMIXIMGPROC_PASSIMAGE_INPUT,\
            rmiximgproc_passimage_call)
252
254 /** \def RMIXIMGPROC_PASSIMAGE_METHOD_CLIENT
    * \brief Client-side descriptor for pass image method.
256 */
    #define RMIXIMGPROC_PASSIMAGE_METHOD_CLIENT\
258     RMIX_METHOD_INIT("passimage",\
        RMIXIMGPROC_PASSIMAGE_OUTPUT,\
260         RMIXIMGPROC_PASSIMAGE_INPUT,\
            NULL)
262
264 /*****
    *
266 * Method descriptor for set image counter method.
    *
268 *****/
270 /** \def RMIXIMGPROC_SETIMAGECOUNTER_INPUT
    * \brief Input signature for set image counter method.
272 */
    #define RMIXIMGPROC_SETIMAGECOUNTER_INPUT\
274     RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(unsigned_int,))    /* counter */
276
278 /** \def RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT
    * \brief Output signature for set image counter method.
280 */
    #define RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT\
282     RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))          /* return */
284
286 /** \def RMIXIMGPROC_SETIMAGECOUNTER_METHOD
    * \brief Server-side descriptor for set image counter method.
288 */
    #define RMIXIMGPROC_SETIMAGECOUNTER_METHOD\
290     RMIX_METHOD_INIT("setimagecounter",\
        RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT,\
292         RMIXIMGPROC_SETIMAGECOUNTER_INPUT,\
            rmiximgproc_setimagecounter_call)
294
296 /** \def RMIXIMGPROC_SETIMAGECOUNTER_METHOD_CLIENT
```

A. Appendix

```

    * \brief Client-side descriptor for set image counter method.
298 */
#define RMXIMGPROC_SETIMAGECOUNTER_METHOD_CLIENT\
300     RMX_METHOD_INIT("setimagecounter",\
                       RMXIMGPROC_SETIMAGECOUNTER_OUTPUT,\
302                       RMXIMGPROC_SETIMAGECOUNTER_INPUT,\
                       NULL)
304

306 /*****
    *
308 * Method descriptor for update image counter method.
    *
310 *****/

312 /** \def  RMXIMGPROC_UPDATEIMAGECOUNTER_INPUT
    * \brief Input signature for update image counter method.
314 */
#define RMXIMGPROC_UPDATEIMAGECOUNTER_INPUT\
316     RMX_SIGNATURE(\
        RMX_SIGNATURE_TYPE(unsigned_int,))          /* counter */
318

320 /** \def  RMXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT
    * \brief Output signature for update image counter method.
322 */
#define RMXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT\
324     RMX_SIGNATURE(\
        RMX_SIGNATURE_TYPE(int,))                  /* return */
326

328 /** \def  RMXIMGPROC_UPDATEIMAGECOUNTER_METHOD
    * \brief Server-side descriptor for update image counter method.
330 */
#define RMXIMGPROC_UPDATEIMAGECOUNTER_METHOD\
332     RMX_METHOD_INIT("updateimagecounter",\
                       RMXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT,\
334                       RMXIMGPROC_UPDATEIMAGECOUNTER_INPUT,\
                       rmiximgproc_setimagecounter_call)
336

338 /** \def  RMXIMGPROC_UPDATEIMAGECOUNTER_METHOD_CLIENT
    * \brief Client-side descriptor for update image counter method.
340 */
#define RMXIMGPROC_UPDATEIMAGECOUNTER_METHOD_CLIENT\
342     RMX_METHOD_INIT("updateimagecounter",\
                       RMXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT,\
344                       RMXIMGPROC_UPDATEIMAGECOUNTER_INPUT,\
                       NULL)
346

348 /*****
    *
350 * Method descriptor for update predecessor method.
    *
352 *****/

354 /** \def  RMXIMGPROC_UPDATEPREDECESSOR_INPUT
    * \brief Input signature for update predecessor method.
356 */
#define RMXIMGPROC_UPDATEPREDECESSOR_INPUT\
358     RMX_SIGNATURE(\
        RMX_SIGNATURE_TYPE(string, ))              /* successor */

```

A. Appendix

```
360
362 /** \def RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT
    * \brief Output signature for update predecessor method.
364 */
365 #define RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT\
366     RMIX_SIGNATURE(\
367         RMIX_SIGNATURE_TYPE(int , ) /* return */\
368         RMIX_SIGNATURE_TYPE(unsigned_int , )) /* image counter */
370
371 /** \def RMIXIMGPROC_UPDATEPREDECESSOR_METHOD
372 * \brief Server-side descriptor for update predecessor method.
    */
373 #define RMIXIMGPROC_UPDATEPREDECESSOR_METHOD\
374     RMIX_METHOD_INIT("updatepredecessor",\
375         RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT,\
376         RMIXIMGPROC_UPDATEPREDECESSOR_INPUT,\
377         rmiximgproc_updatepredecessor_call)
380
381 /** \def RMIXIMGPROC_UPDATEPREDECESSOR_METHOD_CLIENT
382 * \brief Client-side descriptor for update predecessor method.
    */
383 #define RMIXIMGPROC_UPDATEPREDECESSOR_METHOD_CLIENT\
384     RMIX_METHOD_INIT("updatepredecessor",\
385         RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT,\
386         RMIXIMGPROC_UPDATEPREDECESSOR_INPUT,\
387         NULL)
390
391 /*****
392 *
393 * Method descriptor for update successor method.
394 *
395 *****/
396
397 /** \def RMIXIMGPROC_UPDATESUCCESSOR_INPUT
398 * \brief Input signature for update successor method.
    */
399 #define RMIXIMGPROC_UPDATESUCCESSOR_INPUT\
400     RMIX_SIGNATURE(\
401         RMIX_SIGNATURE_TYPE(string , ) /* predecessor */\
402         )
404
405 /** \def RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT
406 * \brief Output signature for update successor method.
    */
407 #define RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT\
408     RMIX_SIGNATURE(\
409         RMIX_SIGNATURE_TYPE(int , ) /* return */\
410         )
412
413 /** \def RMIXIMGPROC_UPDATESUCCESSOR_METHOD
414 * \brief Server-side descriptor for update successor method.
    */
415 #define RMIXIMGPROC_UPDATESUCCESSOR_METHOD\
416     RMIX_METHOD_INIT("updatesuccessor",\
417         RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT,\
418         RMIXIMGPROC_UPDATESUCCESSOR_INPUT,\
419         rmiximgproc_updatesuccessor_call)
422
```

A. Appendix

```
/** \def RMIXIMGPROC_UPDATESUCCESSOR_METHOD_CLIENT
424 * \brief Client-side descriptor for update successor method.
*/
426 #define RMIXIMGPROC_UPDATESUCCESSOR_METHOD_CLIENT\
    RMIX_METHOD_INIT("updatesuccessor",\
428                 RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT,\
                 RMIXIMGPROC_UPDATESUCCESSOR_INPUT,\
430                 NULL)

432
433 /*****
434 *
435 * Method descriptor for imageprocessed.
436 *
437 *****/
438
439 /** \def RMIXIMGPROC_IMAGEPROCESSED_INPUT
440 * \brief Input signature for image processed method.
441 */
442 #define RMIXIMGPROC_IMAGEPROCESSED_INPUT\
    RMIX_SIGNATURE(\
444         RMIX_SIGNATURE_TYPE(string , ))          /* image name */

446
447 /** \def RMIXIMGPROC_IMAGEPROCESSED_OUTPUT
448 * \brief Output signature for image processed method.
449 */
450 #define RMIXIMGPROC_IMAGEPROCESSED_OUTPUT\
    RMIX_SIGNATURE(\
452         RMIX_SIGNATURE_TYPE(int ,))          /* return */

454
455 /** \def RMIXIMGPROC_IMAGEPROCESSED_METHOD
456 * \brief Server-side descriptor for image processed method.
457 */
458 #define RMIXIMGPROC_IMAGEPROCESSED_METHOD\
    RMIX_METHOD_INIT("imageprocessed",\
460                 RMIXIMGPROC_IMAGEPROCESSED_OUTPUT,\
                 RMIXIMGPROC_IMAGEPROCESSED_INPUT,\
462                 rmiximgproc_imageprocessed_call)

464
465 /** \def RMIXIMGPROC_IMAGEPROCESSED_METHOD_CLIENT
466 * \brief Client-side descriptor for image processed method.
467 */
468 #define RMIXIMGPROC_IMAGEPROCESSED_METHOD_CLIENT\
    RMIX_METHOD_INIT("imageprocessed",\
470                 RMIXIMGPROC_IMAGEPROCESSED_OUTPUT,\
                 RMIXIMGPROC_IMAGEPROCESSED_INPUT,\
472                 NULL)

474
475 /*****
476 *
477 * Interface for rmix imgproc.
478 *
479 *****/
480
481 /** \def RMIXIMGPROC_METHODS
482 * \brief Server-side method descriptors.
483 */
484 #define RMIXIMGPROC_METHODS \
    { \
```

A. Appendix

```
486     RMXIMGPROC_INITPIPELINE_METHOD,          /* init pipeline method */
487     RMXIMGPROC_INVOKEPIPELINE_METHOD,        /* invoke pipeline method */
488     RMXIMGPROC_PASSIMAGE_METHOD,             /* pass image method */
489     RMXIMGPROC_SETIMAGECOUNTER_METHOD,        /* set image counter method */
490     RMXIMGPROC_AVAILABILITYCHECK_METHOD,      /* check plug-in existence */
491     RMXIMGPROC_IMAGEPROCESSED_METHOD,        /* image processed */
492     RMXIMGPROC_UPDATEPREDECESSOR_METHOD,      /* update predecessor */
493     RMXIMGPROC_UPDATESUCCESSOR_METHOD,        /* update successor */
494     RMXIMGPROC_SETIMAGECOUNTER_METHOD,        /* update image counter */
495     RMXIMGPROC_SENDWORKLIST_METHOD           /* send work list method */
496 }
497
498 /** \def RMXIMGPROC_METHODS_CLIENT
499 * \brief Client-side method descriptors.
500 */
501 #define RMXIMGPROC_METHODS_CLIENT \
502 { \
503     RMXIMGPROC_INITPIPELINE_METHOD_CLIENT,    /* init pipeline method */
504     RMXIMGPROC_INVOKEPIPELINE_METHOD_CLIENT,  /* invoke pipeline method */
505     RMXIMGPROC_PASSIMAGE_METHOD_CLIENT,       /* pass image method */
506     RMXIMGPROC_SETIMAGECOUNTER_METHOD_CLIENT, /* set iamge count method */
507     RMXIMGPROC_AVAILABILITYCHECK_METHOD_CLIENT, /* check existence */
508     RMXIMGPROC_IMAGEPROCESSED_METHOD_CLIENT, /* image processed */
509     RMXIMGPROC_UPDATEPREDECESSOR_METHOD_CLIENT, /* update predecessor */
510     RMXIMGPROC_UPDATESUCCESSOR_METHOD_CLIENT, /* update successor */
511     RMXIMGPROC_UPDATEIMAGECOUNTER_METHOD_CLIENT, /* set iamge counter */
512     RMXIMGPROC_SENDWORKLIST_METHOD_CLIENT     /* send work list */
513 }
514
515 /** \def RMXIMGPROC_METHODS_INITPIPELINE_INDEX
516 * \brief Method indice initpipeline.
517 */
518 #define RMXIMGPROC_METHODS_INITPIPELINE_INDEX 0 /* init pipe index */
519
520 /** \def RMXIMGPROC_METHODS_INVOKEPIPELINE_INDEX
521 * \brief Method indice invokepipeline.
522 */
523 #define RMXIMGPROC_METHODS_INVOKEPIPELINE_INDEX 1 /* invoke pipe index */
524
525 /** \def RMXIMGPROC_METHODS_PASSIMAGE_INDEX
526 * \brief Method indice passimage.
527 */
528 #define RMXIMGPROC_METHODS_PASSIMAGE_INDEX 2 /* pass image index */
529
530 /** \def RMXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX
531 * \brief Method indice setimagecounter.
532 */
533 #define RMXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX 3 /* set image counter */
534
535 /** \def RMXIMGPROC_METHODS_AVAILABILITYCHECK_INDEX
536 * \brief Method indice availabilitycheck.
537 */
538 #define RMXIMGPROC_METHODS_AVAILABILITYCHECK_INDEX 4 /* check existence */
539
540 /** \def RMXIMGPROC_METHODS_IMAGEPROCESSED_INDEX
541 * \brief Method indice imageprocessed.
542 */
543 #define RMXIMGPROC_METHODS_IMAGEPROCESSED_INDEX 5 /* image processed */
544
545 /** \def RMXIMGPROC_METHODS_UPDATEPREDECESSOR_INDEX
546 * \brief Method indice update predecessor.
547 */
```


A. Appendix

```
550 */
550 #define RMIXIMGPROC_METHODS_UPDATEPREDECESSOR_INDEX 6 /* updatepredecessor */
552 /** \def RMIXIMGPROC_METHODS_UPDATESUCCESSOR_INDEX
552 * \brief Method indice update successor.
554 */
554 #define RMIXIMGPROC_METHODS_UPDATESUCCESSOR_INDEX 7 /* update successor */
556
556 /** \def RMIXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX
558 * \brief Method indice updateimagecounter.
558 */
560 #define RMIXIMGPROC_METHODS_UPDATEIMAGECOUNTER_INDEX 8 /* update counter */
562
562 /** \def RMIXIMGPROC_METHODS_SENDWORKLIST_INDEX
562 * \brief Method indice send worklist.
564 */
564 #define RMIXIMGPROC_METHODS_SENDWORKLIST_INDEX 9 /* send worklist */
566
566 /** \def RMIXIMGPROC_METHODS_COUNT
568 * \brief Method indices for rmix image processing methods.
568 */
570 #define RMIXIMGPROC_METHODS_COUNT 10 /* method count */
572
572 /*****
574 *
574 * Includes
576 *
576 *****/
578
578 /* Include <harness-rmix/harness-rmix.h> header. */
580 #if HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
580 #include <harness-rmix/harness-rmix.h>
582 #endif
584
584 /*****
586 *
586 * Data
588 *
588 *****/
590
590 /** \var extern const rmix_method_t \
592 rmiximgproc_methods[RMIXIMGPROC_METHODS_COUNT];
592 * \brief Server-side method descriptors for rmix imgproc.
594 */
594 extern const rmix_method_t rmiximgproc_methods[RMIXIMGPROC_METHODS_COUNT];
596
596
598 /** \var extern const rmix_interface_t rmiximgproc_interface;
598 * \brief Server-side interface for rmix imgproc.
600 */
600 extern const rmix_interface_t rmiximgproc_interface;
602
602
604 #endif /* RMIXIMGPROC_RMIXIMGPROC_H */
606 /*****
608 *
608 * END OF FILE
610 *
610 *****/
```

A.3.2. Parallel Plug-in Manager

A.3.2.1. Header File for the PPM

```

/* libppm/ppm.h.  Generated by configure.  */
2 /*****
*
4 * Header file for the parallel plug-in manager module.
* Copyright (c) Ronald Baumann
6 *
* For more information see the following files in the source distribution top-
8 * level directory or package data directory (usually /usr/local/share/package):
*
10 * - README    for general package information.
* - INSTALL   for package install information.
12 * - COPYING   for package license information and copying conditions.
* - AUTHORS   for package authors information.
14 * - ChangeLog for package changes information.
*
16 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
* top-level directory to create the target file.
18 *
*****/
20
/** \file ppm.h
22 * \brief Header file for parallel plug-in manager module.
*
24 * The libppm module is a plug-in for the Harness project. It reads a
* configuration file and starts a parallel plug-in on a defined number
26 * of harness daemons.
*/
28
/* Avoid to include the content of this header file twice. */
30 #ifndef PPM_PPM_H
#define PPM_PPM_H
32

34 /*****
*
36 * Macros
*
38 *****/

40 /** \def LIBPPM_PACKAGE_NAME
* \brief Package name (unquoted).
42 */
#define LIBPPM_PACKAGE_NAME PPM
44

46 /** \def LIBPPM_PACKAGE_VERSION
* \brief Package version (unquoted).
48 */
#define LIBPPM_PACKAGE_VERSION 1.1
50

52 /** \def LIBPPM_PACKAGE_RELEASE
* \brief Package release (unquoted).
54 */
#define LIBPPM_PACKAGE_RELEASE 0
56

```

A. Appendix

```
58 /** \def LIBPPM_PACKAGE_BUGREPORT
   * \brief Package bug report e-mail (unquoted).
60 */
#define LIBPPM_PACKAGE_BUGREPORT baumannr@ornl.gov
62

64 /** \def LIBPPM_VERSION_CURRENT
   * \brief Version current (unquoted).
66 */
#define LIBPPM_VERSION_CURRENT 0
68

70 /** \def LIBPPM_VERSION_REVISION
   * \brief Version revision (unquoted).
72 */
#define LIBPPM_VERSION_REVISION 0
74

76 /** \def LIBPPM_VERSION_AGE
   * \brief Version age (unquoted).
78 */
#define LIBPPM_VERSION_AGE 0
80

82 /** \def LIBPPM_VERSION_FIRST
   * \brief Version first (unquoted).
84 */
#define LIBPPM_VERSION_FIRST 0
86

88 /** \def LIBPPM_VERSION
   * \brief Version (unquoted).
90 */
#define LIBPPM_VERSION 0.0.0
92

94 /** \def PPM_QUOTES(string)
   * \brief Quoting a string.
96 */
#define PPM_QUOTES(string) #string
98

100 /** \def PPM_STRING(string)
   * \brief A string.
102 */
#define PPM_STRING(string) PPM_QUOTES(string)
104

106 /** \def PPM_FUSE(arg1, arg2)
   * \brief Fuse two strings.
108 */
#define PPM_FUSE(arg1, arg2) arg1##arg2
110

112 /** \def PPM_JOIN(arg1, arg2)
   * \brief Joining two text constants.
114 */
#define PPM_JOIN(arg1, arg2) PPM_FUSE(arg1, arg2)
116

118 /** \def PPM_WARN(string)
   * \brief Debug printout.
120 */
```

A. Appendix

```
122 #define PPM_WARN(string) \  
123     fprintf(stderr,\  
124             PPM_STRING(warn:libppm:%d:%s:%u:%s: string\n),\  
             getpid(), __FILE__, __LINE__, __FUNC__\  
126 \  
127 /** \def PPM_INFO(string)  
128 * \brief Debug printout.  
129 */  
130 #define PPM_INFO(string) \  
131     fprintf(stderr,\  
132             PPM_STRING(info:libppm:%d:%s:%u:%s: string\n),\  
             getpid(), __FILE__, __LINE__, __FUNC__\  
134 \  
135 /** \def PPM_PRINT(string)  
136 * \brief Wrapper for debug printout.  
137 */  
138 #ifdef DEBUG  
139 #define PPM_PRINT(string) string);  
140 #else /* DEBUG */  
141 #define PPM_PRINT(string)  
142 #endif /* DEBUG */  
144 \  
145 /* Flag for <harness.0/harness.h> header. */  
146 #ifndef HARNESS_HARNESS_H  
147 #define HARNESS_HARNESS_H  
148 #define HAVE_HARNESS_0_HARNESS_H 1  
149 #endif /* HARNESS_HARNESS_H */  
150 #endif /* HARNESS_HARNESS_H */  
152 \  
153 /** \def FREE(x)  
154 * \brief Macro frees allocated memory and add the NULL pointer.  
155 *  
156 * First the pointer is checked whether it is not NULL. Then it is freed.  
157 */  
158 #define FREE(x) {if (x != NULL) {free(x); x = NULL;}}  
160 \  
161 /** \def SOURCEDIR  
162 * \brief Dirs entry in the imgproc input file.  
163 */  
164 #define SOURCEDIR "sourcedir"  
166 \  
167 /** \def TARGETDIR  
168 * \brief Target directory entry in the imgproc input file.  
169 */  
170 #define TARGETDIR "targetdir"  
172 \  
173 /*****  
174 *  
175 * Includes  
176 *  
177 *****/  
179 /* Include <harness.0/harness.h> header. */  
180 #if HAVE_HARNESS_0_HARNESS_H  
181 #include <harness.0/harness.h>  
182 #endif /* HAVE_HARNESS_0_HARNESS_H */
```

A. Appendix

```
184 #include "readconf.h"
186 #include "rmixintegral.h"
187 #include "rmiximgproc.h"
188 #include "rmixppm.h"

190
191 /*****
192  *
193  * Data Types
194  *
195  *****/
196
197 /** \struct ppm_t
198  * \brief Parallel Plug-in Manager "class".
199  *
200  * This structure contains the public plug-in data and function pointers.
201  * It also provides information about the version of the plug-in.
202  */
203 typedef struct
204 {
205     struct
206     {
207         const unsigned int current; /* current version */
208         const unsigned int revision; /* current revision */
209         const unsigned int age; /* version age */
210         const unsigned int first; /* first supported version */
211     } version; /* plug-in version */
212 } ppm_t;

214
215 /*****
216  *
217  * Function Prototypes
218  *
219  *****/
220
221 /** \fn int ppm_init_rmix();
222  * \brief Exports and initialises the RMX library.
223  *
224  * \return 0 on success or -1 on any error
225  */
226 int ppm_init_rmix();

228
229 /** \fn int ppm_fini_rmix();
230  * \brief Finalizes and unexports RMX.
231  *
232  * \return 0 on success or -1 on any error
233  */
234 int ppm_fini_rmix();

236
237 /** \fn int ppm_start()
238  * \brief Invokes the Parallel Plug-in Manager execution.
239  *
240  * \return 0 on success or -1 on any error
241  */
242 int ppm_start();

244
245 /** \fn int ppm_load_harnesskernel_replicated( configlist_t *nodelist, \
246                                               configlist_t *pluginlist);
```

A. Appendix

```

248 * \brief Loads a replicated parallel plug-in on the Harness kernels specified
    * in the conf file.
    *
250 * \param *nodelist Name of the node(s)
    * \param *pluginlist Name of the parallel plug-in
252 * \return 0 on success or -1 on any error
    */
254 int ppm_load_harnesskernel_replicated( configlist_t *nodelist,
                                         configlist_t *pluginlist);
256
258 /** \fn int ppm_load_harnesskernel_distributed( configlist_t *nodelist, \
                                                configlist_t *pluginlist, \
260                                                char *ppmnode);
    * \brief Loads a distributed parallel plug-in on the Harness kernels
262 * specified in the conf file.
    *
264 * \param **nodelist Name of the node(s)
    * \param *pluginlist Name of the parallel plug-in
266 * \param *ppmnode Node running the PPM
    * \return 0 on success or -1 on any error
268 */
270 int ppm_load_harnesskernel_distributed( configlist_t **nodelist,
                                         configlist_t *pluginlist,
                                         char *ppmnode);
272
274 /** \fn int ppm_unload_harnesskernel( configlist_t *nodelist);
    * \brief Unloads the Harness kernels on the nodes specified in the conf file.
276 * The kernels are completely killed inclusive all running plug-ins.
    * Careful use of function is advised.
278
    * \param *nodelist Name of the node(s)
280 * \return 0 on success or -1 on any error
    */
282 int ppm_unload_harnesskernel( configlist_t *nodelist);
284
286 /** \fn int ppm_unload_harnesskernel_soft( configlist_t *nodelist);
    * \brief Unloads the Harness kernels on the nodes specified in the conf file.
288 * The kernels are completely killed inclusive all running plug-ins.
    * For terminating the kernel the internal shutdown function is used.
290
    * \param *nodelist Name of the node(s)
    * \return 0 on success or -1 on any error
292 */
294 int ppm_unload_harnesskernel_soft( configlist_t *nodelist);
296
298 /** \fn void ppm_sleep(int sec_dlay, int usec_dlay);
    * \brief The process sleeps a preset time interval.
300
    * \param sec_dlay Delay in seconds
    * \param usec_dlay Delay in micro seconds
    */
302 void ppm_sleep(int sec_dlay, int usec_dlay);
304
306 /** \fn int ppm_createremoteref( rmix_remoteref_t **remoteobj,\
                                char *objectid,\
                                char *node);
308 * \brief Creates a remote reference specified by objectid and node. The RPC
    * protocol is used.
```

A. Appendix

```

310 *
311 * \param **remoteobj ID of the remote object
312 * \param *objectid ID of the exported object
313 * \param *node Name of the node
314 * \return 0 on success or -1 on any error
315 */
316 int ppm_createremoteref( rmix_remoteref_t **remoteobj,
317                          char *objectid,
318                          char *node);
319
320
321 /*****
322 *
323 * Monte Carlo section
324 *
325 *****/
326
327 /** \fn int ppm_monte_carlo_loader( configlist_t *nodelist, \
328                                     configlist_t *replicatedlist, \
329                                     configlist_t *inputlist)
330 * \brief Invokes the Monte Carlo parallel plug-in execution.
331 *
332 * \param *nodelist Name of the node(s)
333 * \param *replicatedlist Name of replicated parallel plug-in
334 * \param *inputlist List with name(s) of input file(s)
335 * \return 0 on success or -1 on any error
336 */
337 int ppm_monte_carlo_loader( configlist_t *nodelist,
338                             configlist_t *replicatedlist,
339                             configlist_t *inputlist);
340
341
342 /** \fn int rmixintegralclient_integration_send( \
343                                             rmix_invokeref_t **invokeref, \
344                                             rmix_remoteref_t *remoteobject, \
345                                             unsigned int iterations, \
346                                             double lowerbound, \
347                                             double upperbound, \
348                                             unsigned int numcoeffs, \
349                                             double *coeffs);
350 * \brief Sends an integral dataset to a server (client-side method stub).
351 *
352 * \param **invokeref The invocation reference (also return).
353 * \param *remoteobject The remote object reference.
354 * \param iterations The number of the generated random numbers.
355 * \param lowerbound The lower boundary of the integration interval.
356 * \param upperbound The upper boundary of the integration interval.
357 * \param numcoeffs The number of function coefficients.
358 * \param *coeffs The coefficients.
359 * \return 0 on success or -1 on any error.
360 */
361 int rmixintegralclient_integration_send( rmix_invokeref_t **invokeref,
362                                         rmix_remoteref_t *remoteobject,
363                                         unsigned int iterations,
364                                         double lowerbound,
365                                         double upperbound,
366                                         unsigned int numcoeffs,
367                                         double *coeffs);
368
369
370 /** \fn int rmixintegralclient_integration_retrieve( \
371                                             rmix_invokeref_t **invokeref, \
372                                             double *integral);

```

A. Appendix

```

374 * \brief Retrieves rmixintegral_integration invocation result (client-side
* method stub).
*
376 * Returns the output of a previous asynchronous invocation that sends the
* integration parameters to the server and starts the integration
378 * process.
*
380 * \param **invokeref The invocation reference.
* \param *integral The result of the integration.
382 * \return 0 on success or -1 on any error.
*/
384 int rmixintegralclient_integration_retrieve( rmix_invokeref_t **invokeref,
double *integral);
386
388
/** \fn int ppm_read_integralinput( configlist_t *inputlist, \
390 unsigned int *number, \
double **coeffs, \
392 int *iterations, \
double *lower, \
394 double *upper);
* \brief Reads the input file, which contains the needed information for
396 * the integral computation.
*
398 * \param *inputlist List with name(s) of input file(s)
* \param *number Returns the number of coefficients
400 * \param **coeffs Returns the array of coefficients
* \param *iterations Returns the amount of random numbers, which are used
402 * \param *lower Returns the lower boundary of the interval
* \param *upper Returns the upper boundary of the interval
404 * \return 0 on success or -1 on any error
*/
406 int ppm_read_integralinput( configlist_t *inputlist,
unsigned int *number,
408 double **coeffs,
int *iterations,
410 double *lower,
double *upper);
412
414 /** \fn int ppm_schedule_integral( configlist_t *nodelist, \
configlist_t *pluginlist, \
416 unsigned int numcoeffs, \
double *coeffs, \
418 int iterations, \
double lower_border, \
420 double upper_border);
* \brief Partitions the integration interval and sends it to the parallel
422 * plug-in units.
*
424 * \param *nodelist Name of the node(s)
* \param *pluginlist Name of the parallel plug-in
426 * \param numcoeffs Number of coefficients
* \param *coeffs Array of coefficients
428 * \param iterations Amount of random numbers, which are used
* \param lower_border Lower boundary of the interval
430 * \param upper_border Upper boundary of the interval
* \return 0 on success or -1 on any error
432 */
int ppm_schedule_integral( configlist_t *nodelist,
434 configlist_t *plugin_list,
unsigned int numcoeffs,
```


A. Appendix

```
436         double    *coeffs ,
437         int        iterations ,
438         double    lower_border ,
439         double    upper_border );
440
441 /*****
442 *
443 * Image Processing section
444 *
445 *****/
446
447 /** \fn      int rmiximgprocclient_initpipeline (rmix_remoteref_t *remoteobj, \
448                                               unsigned int    filter, \
449                                               char            *sourcedir, \
450                                               char            *targetdir, \
451                                               char            *successor, \
452                                               char            *predecessor, \
453                                               char            *ppmnode);
454 * \brief    Initialises the pipeline (client-side method stub).
455 *
456 * \param   *remoteobj  ID of the remote object
457 * \param   filter      Image filter which will be used
458 * \param   *sourcedir  Directory containing image sources.
459 * \param   *targetdir  Directory containing processed images
460 * \param   *successor  ID of the next plug-in in the pipeline
461 * \param   *predecessor ID of the previous plug-in in the pipeline
462 * \param   *ppmnode    Node, on which the PPM runs
463 * \return  0 on success or -1 on any error
464 */
465 int rmiximgprocclient_initpipeline ( rmix_remoteref_t *remoteobj,
466                                     unsigned int    filter,
467                                     char            *sourcedir,
468                                     char            *targetdir,
469                                     char            *successor,
470                                     char            *predecessor,
471                                     char            *ppmnode);
472
473
474 /** \fn      int rmiximgprocclient_invokepipeline_oneway ( \
475                                               rmix_remoteref_t *remoteobj);
476 * \brief    Invokes the pipeline processing (client-side method stub).
477 *
478 * \param   *remoteobj  ID of the remote object
479 * \return  0 on success or -1 on any error
480 */
481 int rmiximgprocclient_invokepipeline_oneway (rmix_remoteref_t *remoteobj);
482
483
484 /** \fn      int rmiximgprocclient_availabilitycheck( \
485                                               rmix_remoteref_t *remoteobj);
486 * \brief    Checks the availability of a plug-in (client-side method stub).
487 *
488 * \param   *remoteobj  ID of the remote object
489 * \return  0 on success or -1 on any error
490 */
491 int rmiximgprocclient_availabilitycheck( rmix_remoteref_t *remoteobj);
492
493
494
495 /** \fn      int ppm_imageprocessing_loader( configlist_t *nodelist, \
496                                               configlist_t *distributedlist, \
497                                               configlist_t *inputlist);
```

A. Appendix

```

500 * \brief Invokes the image processing pipeline handling of the PPM.
501 *
502 * \param *nodelist      Name of the node(s)
503 * \param *distributedlist  Name of replicated parallel plug-in
504 * \param *inputlist     List with name(s) of input file(s)
505 * \return              0 on success or -1 on any error
506 */
507 int ppm_imageprocessing_loader( configlist_t *nodelist ,
508                               configlist_t *distributedlist ,
509                               configlist_t *inputlist);
510
511 /** \fn      int ppm_read_imageinput( configlist_t *inputlist , \
512                                     char          **sourcedir , \
513                                     char          **targetdir);
514 * \brief Reads the input file , which contains the needed information for
515 * image processing.
516 *
517 * \param *inputlist     List with name(s) of input file(s)
518 * \param **sourcedir    Returns the source directory with images
519 * \param **targetdir    Returns target directory
520 * \return              0 on success or -1 on any error
521 */
522 int ppm_read_imageinput( configlist_t *inputlist ,
523                          char          **sourcedir ,
524                          char          **targetdir);
525
526 /** \fn      int rmiximgprocclient_updateimagecounter( \
527                                     rmix_remoteref_t *remoteobj,\
528                                     unsigned int      imagecounter);
529 * \brief Updates the image counter a the plug-in (client-side method stub).
530 *
531 * \param *remoteobj     ID of the remote object
532 * \param imagecounter   New image counter value
533 * \return              0 on success or -1 on any error
534 */
535 int rmiximgprocclient_updateimagecounter( rmix_remoteref_t *remoteobj ,
536                                           unsigned int      imagecounter);
537
538 /** \fn      int rmiximgprocclient_updatepredecessor( \
539                                     rmix_remoteref_t *remoteobj , \
540                                     char          *successor , \
541                                     unsigned int      imagecounter);
542 * \brief Updates the successor entry of a predecessor plug-in (client-side
543 * method stub).
544 *
545 * \param *remoteobj     ID of the remote object
546 * \param *successor     New successor entry
547 * \param *imagecounter  The returned image counter from the predecessor
548 * \return              0 on success or -1 on any error
549 */
550 int rmiximgprocclient_updatepredecessor( rmix_remoteref_t *remoteobj ,
551                                           char          *successor ,
552                                           unsigned int      imagecounter);
553
554 /** \fn      int rmiximgprocclient_updatesuccessor( \
555                                     rmix_remoteref_t *remoteobj , \
556                                     char          *predecessor);
557 * \brief Updates the predecessor entry of a successor plug-in (client-side
558 * method stub).
```

A. Appendix

```
562 *
563 * \param *remoteobj ID of the remote object
564 * \param *predecessor New predecessor entry
565 * \return 0 on success or -1 on any error
566 */
567 int rmiximgprocclient_updatesuccessor( rmix_remoteref_t *remoteobj,
568                                     char *predecessor );
569
570
571 /** \fn RMX_METHOD_CALL(rmixppm_repairpipe_call);
572 * \brief Accepts calls for repairing the pipeline in case of an error.
573 * (server-side stub)
574 *
575 * \param object The local object.
576 * \param outary The output values array. The values array and its
577 * containing values are allocated before the call with the
578 * exception of variable arrays and strings. They are allocated
579 * dynamically or explicitly set to NULL by this function. In
580 * the case of variable arrays, the length value is allocated
581 * before the call and is set to 0 if its variable array is
582 * NULL.
583 * \param outcnt The output values count.
584 * \param inary The input values array. The values array and its containing
585 * values are not modified or deallocated by this call.
586 * \param incnt The input values count.
587 * \return 0 on success or -1 on any error.
588 */
589 RMX_METHOD_CALL(rmixppm_repairpipe_call);
590
591 /** \fn int rmixppm_repairpipe (char *node);
592 * \brief Tries to repair the pipeline in case of an error.
593 *
594 * \param *node Node which cannot be accessed.
595 * \return 0 on success or -1 on any error.
596 */
597 int rmixppm_repairpipe (char *node);
598
599
600
601 /** \fn int rmiximgprocclient_sendworklist_oneway(
602                                     rmix_remoteref_t *remoteobj);
603 * \brief Triggers a plug-in unit to resend its backup list to its successor
604 * plug-in (client-side method stub).
605 *
606 * \param *remoteobj ID of the remote object
607 * \return 0 on success or -1 on any error
608 */
609 int rmiximgprocclient_sendworklist_oneway (rmix_remoteref_t *remoteobj);
610
611
612
613 /*****
614 * Data
615 *
616 *****/
617
618 /** \var extern ppm_t ppm
619 * \brief Parallel Plug-in Manager "object".
620 *
621 * An external "object" of the Parallel Plug-in Manager "class".
622 */
623 extern ppm_t ppm;
```

A. Appendix

```
626 #endif /* PPM_PPM_H */
628
630 /*****
631  *
632  * END OF FILE
633  *
634  *****/
```

A.3.2.2. Source File for the PPM

```

2  *
3  * Source file for the parallel plug-in manager module.
4  * Copyright (c) Ronald Baumann.
5  *
6  * For more information see the following files in the source distribution top-
7  * level directory or package data directory (usually /usr/local/share/package):
8  *
9  * - README    for general package information.
10 * - INSTALL   for package install information.
11 * - COPYING   for package license information and copying conditions.
12 * - AUTHORS   for package authors information.
13 * - ChangeLog for package changes information.
14 *
15 *****/
16
17 /** \file ppm.c
18 * \brief Source file for the parallel plug-in manager module.
19 *
20 * The libppm module is a plug-in for the Harness project. It reads a
21 * configuration file and starts a parallel plug-in on a defined number
22 * of harness daemons.
23 */
24
25
26 /*****
27  *
28  * Includes
29  *
30  *****/
31
32 /* Main module header file. */
33 #include "ppm.h"
34
35
36 /*****
37  *
38  * Data Types
39  *
40  *****/
41
42 /** \struct ppm_data_t
43 * \brief Parallel Plug-in Manager module data type.
44 *
45 * Contains the mutex and the instances array.
46 */
47 typedef struct
48 {
49     pthread_mutex_t mutex;           /* mutex */
50 }
```

A. Appendix

```
50 pthread_mutex_t pipemutex; /* pipe mutex */
51 unsigned int count; /* instances array count */
52 struct
53 {
54     unsigned int handle; /* instance handle */
55 } *instances; /* instances array */
56 unsigned int handle_rmix; /* Handle for rmix plug-in. */
57 rmix_localref_t *localref_harness; /* exported Harness kernel */
58 rmix_localref_t *localref_ppm; /* exported ppm plug-in */
59 configlist_t *usednodes; /* nodes used for distr. plug-in */
60 configlist_t *freenodes; /* nodes available for distr. plug-in */
61 char *ppmnode; /* node name which runs the PPM */
62 configlist_t *dist_plugins; /* names of the distributed plug-ins */
63 char *sourcedir; /* source directory with images */
64 char *targetdir; /* target directory for images */
65 } ppm_data_t;
66
67
68 /*****
69 *
70 * Function Prototypes
71 *
72 *****/
73
74 /** \fn HARNESS_PLUGINS_INIT(ppm_init);
75 * \brief Initializes the PPM plug-in.
76 *
77 * \param handle The plug-in instance handle.
78 * \return 0 on success or -1 on any error with errno set
79 * appropriately.
80 */
81 HARNESS_PLUGINS_INIT(ppm_init);
82
83
84 /** \fn HARNESS_PLUGINS_FINI(ppm_fini);
85 * \brief Finalizes the PPM plug-in.
86 *
87 * \param handle The plug-in instance handle.
88 * \return 0 on success or -1 on any error with errno set appropriately.
89 */
90 HARNESS_PLUGINS_FINI(ppm_fini);
91
92
93 /*****
94 *
95 * Data
96 *
97 *****/
98
99 /** \var const rmix_method_t rmixppm_methods[RMIXPPM_METHODS_COUNT]
100 * \brief Server-side method descriptors for rmix ppm.
101 */
102 const rmix_method_t rmixppm_methods[RMIXPPM_METHODS_COUNT] =
103     RMIXPPM_METHODS;
104
105
106 /** \var const rmix_interface_t rmixppm_interface
107 * \brief Server-side interface for rmix ppm.
108 */
109 const rmix_interface_t rmixppm_interface =
110 {
111     RMIXPPM_METHODS_COUNT, /* method descriptor count */
112     (rmix_method_t*)rmixppm_methods /* method descriptor array */
113 }
```

A. Appendix

```
};
114

116 /** \var   rmix_method_t rmixintegralclient_methods[RMIXINTEGRAL_METHODS_COUNT]
    * \brief Client-side method descriptors for rmix integral.
118 */
const rmix_method_t rmixintegralclient_methods[RMIXINTEGRAL_METHODS_COUNT] =
120     RMIXINTEGRAL_METHODS_CLIENT;

122
124 /** \var   rmix_interface_t rmixintegralclient_interface
    * \brief Client-side interface for rmix integral.
    */
126 const rmix_interface_t rmixintegralclient_interface = {
    RMIXINTEGRAL_METHODS_COUNT, /* method descriptor count */
128     (rmix_method_t*)rmixintegralclient_methods /* method descriptor array */
};
130

132 /** \var   rmix_method_t rmiximgprocclient_methods[RMIXIMGPROC_METHODS_COUNT]
    * \brief Client-side method descriptors for rmix imgproc.
134 */
const rmix_method_t rmiximgprocclient_methods[RMIXIMGPROC_METHODS_COUNT] =
136     RMIXIMGPROC_METHODS_CLIENT;

138
140 /** \var   const rmix_interface_t rmiximgprocclient_interface
    * \brief Client-side interface for rmix imgproc.
    */
142 const rmix_interface_t rmiximgprocclient_interface = {
    RMIXIMGPROC_METHODS_COUNT, /* method descriptor count */
144     (rmix_method_t*)rmiximgprocclient_methods /* method descriptor array */
};
146

148 /** \var   ppm_data_t ppm_data
    * \brief PPM module data.
    *
150 * Includes the mutex for instances handle and the instances array where the
152 * handles are stored.
    */
154 ppm_data_t ppm_data =
    {
156     PTHREAD_MUTEX_INITIALIZER, /* mutex */
    PTHREAD_MUTEX_INITIALIZER, /* pipe mutex */
158     0, /* instances array count */
    NULL, /* instances array */
160     0, /* handle for rmix plug-in */
    NULL, /* exported Harness kernel */
162     NULL, /* exported ppm plug-in */
    NULL, /* nodes used for distr. plug-in */
164     NULL, /* nodes available for distr. plug-in */
    NULL, /* node name which runs the PPM */
166     NULL, /* names of the distributed plug-ins */
    NULL, /* source directory with images */
168     NULL /* target directory for images */
};
170

172 /** \var   ppm_t ppm
    * \brief Integral plug-in "object".
174 *
    * Contains the version of the library and possible public data and function
```

A. Appendix

```
176 * pointers.
177 */
178 ppm_t ppm =
179 {
180     {
181         LIBPPM_VERSION_CURRENT, /* current version */
182         LIBPPM_VERSION_REVISION, /* current revision */
183         LIBPPM_VERSION_AGE, /* version age */
184         LIBPPM_VERSION_FIRST /* first supported version */
185     } /* plug-in version */
186 };
187
188 /*****
189 *
190 * Functions
191 *
192 *****/
193
194 /*
195 * Initializes the Integral-Loader plug-in.
196 *
197 * handle = The plug-in instance handle.
198 * return = 0 on success or -1 on any error with errno set appropriately.
199 */
200 #undef __FUNC__
201 #define __FUNC__ "ppm_init"
202 HARNESS_PLUGINS_INIT(ppm_init)
203 {
204     int error;
205     void *instances;
206     unsigned int index;
207
208     PPM_PRINT((
209         PPM_INFO(libppm is starting)))
210
211     /* Lock ppm plug-in mutex. */
212     if (0 != (error = pthread_mutex_lock(&ppm_data.mutex)))
213     {
214         errno = error;
215         PPM_PRINT((
216             PPM_WARN(unable to lock ppm plug-in mutex)))
217         harness_syserr();
218         return -1;
219     }
220
221     /* Search for handle in instances array. */
222     for (index = 0; index < ppm_data.count; index++)
223     {
224         if (ppm_data.instances[index].handle == handle)
225         {
226             PPM_PRINT((
227                 PPM_WARN(handle is already in instances array)))
228             harness_syserr();
229
230             /* Unlock integral-loader plug-in mutex. */
231             if (0 != (error =
232                 pthread_mutex_unlock(&ppm_data.mutex)))
233             {
234                 errno = error;
235                 PPM_PRINT((
236                     PPM_WARN(unable to unlock ppm plug-in mutex)))
237                 harness_syserr();
238             }
239         }
240     }
241 }
```

A. Appendix

```
240     }
241     return -1;
242 }
243
244 /* Increase instances array. */
245 index = ppm_data.count;
246 ppm_data.count++;
247 /* Reallocate instances array. */
248 if (NULL == (instances = realloc(ppm_data.instances ,
249                                 ppm_data.count *
250                                 sizeof(ppm_data.instances [0])))
251 {
252     PPM_PRINT((
253         PPM_WARN(unable to reallocate instances array)))
254     /* Unlock integral-loader plug-in mutex. */
255     if (0 != (error = pthread_mutex_unlock(&ppm_data.mutex)))
256     {
257         errno = error;
258         PPM_PRINT((
259             PPM_WARN(unable to unlock ppm plug-in mutex)))
260         harness_syserr();
261     }
262     return -1;
263 }
264
265 ppm_data.instances = instances;
266 /* Save instances array entry. */
267 ppm_data.instances[index].handle = handle;
268 /* Check for first initialization. */
269 if (1 == ppm_data.count)
270 {
271     /******
272     /* PUT YOUR INIT CODE HERE */
273     /******
274     ppm_init_rmix();
275     ppm_start();
276 }
277
278 /* Unlock ppm plug-in mutex. */
279 if (0 != (error = pthread_mutex_unlock(&ppm_data.mutex)))
280 {
281     errno = error;
282     PPM_PRINT((
283         PPM_WARN(unable to unlock ppm plug-in mutex)))
284     harness_syserr();
285     return -1;
286 }
287
288 return 0;
289 }
290
291 /*
292 * Finalizes the parallel plug-in manager.
293 *
294 * handle = The plug-in instance handle.
295 * return = 0 on success or -1 on any error with errno set appropriately.
296 */
297 #undef __FUNC__
298 #define __FUNC__ "ppm_fini"
299 HARNESS_PLUGINS_FINI(ppm_fini)
300 {
```


A. Appendix

```
302     int          error;
303     void          *instances;
304     unsigned int  index;

306     PPM_PRINT((
307         PPM_INFO(libppm is shutting down)))
308
309     /* Lock integral-loader plug-in mutex. */
310     if (0 != (error = pthread_mutex_lock(&ppm_data.mutex)))
311     {
312         errno = error;
313         PPM_PRINT((
314             PPM_WARN(unable to lock ppm plug-in mutex)))
315         harness_syserr();
316         return -1;
317     }
318
319     /* Search for handle in instances array. */
320     for (index = 0; index < ppm_data.count; index++)
321     {
322         if (ppm_data.instances[index].handle == handle)
323         {
324             break;
325         }
326     }
327
328     /* Check if handle is not in instances array. */
329     if (index == ppm_data.count)
330     {
331         PPM_PRINT((
332             PPM_WARN(handle is not in instances array)))
333         harness_syserr();
334         /* Unlock integral-loader plug-in mutex. */
335         if (0 != (error = pthread_mutex_unlock(&ppm_data.mutex)))
336         {
337             errno = error;
338             PPM_PRINT((
339                 PPM_WARN(unable to unlock ppm plug-in mutex)))
340             harness_syserr();
341         }
342         return -1;
343     }
344
345     /* Remove instance from instances array. */
346     ppm_data.count--;
347     memmove(ppm_data.instances + index,
348             ppm_data.instances + index + 1,
349             (ppm_data.count - index) *
350             sizeof(unsigned int));
351
352     /* Reallocate instances array. */
353     if (0 == ppm_data.count)
354     {
355         free(ppm_data.instances);
356         ppm_data.instances = NULL;
357     }
358     else if (NULL == (instances = realloc(ppm_data.instances,
359                                           ppm_data.count *
360                                           sizeof(ppm_data.instances[0])))
361     {
362         PPM_PRINT((
363             PPM_WARN(unable to reallocate instances array)))
364     }
```

A. Appendix

```
else
366 {
    ppm_data.instances = instances;
368 }
/* Check for last finalization. */
370 if (0 == ppm_data.count)
    {
372     /******
        /* PUT YOUR FINI CODE HERE */
374     /******
        ppm_fini_rmix();
376     configlist_delete_list(&ppm_data.usednodes);
        configlist_delete_list(&ppm_data.freenodes);
378     configlist_delete_list(&ppm_data.dist_plugins);
        FREE(ppm_data.sourcedir);
380     FREE(ppm_data.targetdir);
    }
382 }

384 /* Unlock ppm plug-in mutex. */
    if (0 != (error = pthread_mutex_unlock(&ppm_data.mutex)))
386 {
        errno = error;
388     PPM_PRINT((
        PPM_WARN(unable to unlock ppm plug-in mutex))
390     harness_syserr();
        return -1;
392 }
    return 0;
394 }

396
/*
398 * Loads the RMIX library and initializes the necessary rmix parameters.
    *
400 * \return 0 on success or -1 on any error
    */
402 #undef __FUNC__
    #define __FUNC__ "ppm_init_rmix"
404 int ppm_init_rmix()
    {
406     int ret;

408     PPM_PRINT((
        PPM_INFO(start initialization of RMIX)))
410
    /* Load RMIX plug-in and initialize RMIX */
412     ret = harness_plugins_load( &ppm_data.handle_rmix, "libharness-rmix",
        HARNESS_PLUGINS_EXPORT);
414     if (ret != 0)
        {
416         PPM_PRINT((
            PPM_WARN(unable to load RMIX plug-in))
418         return -1;
        }
420
    /* Exports Harness while forcing 1000 as specific object handle */
422     ret = harness_rmix_export4( &ppm_data.localref_harness ,
        "PROTOCOL=RPC OBJECTID=1000");
424     if (ret != 0)
        {
426         PPM_PRINT((
            PPM_WARN(unable to export Harness kernel)))
```

A. Appendix

```
428     return -1;
429 }
430
431 /* Exports ppm plug-in while forcing 1003 as specific object handle */
432 ret = rmix_export4( &ppm_data.localref_ppm ,
433                   "PROTOCOL=RPC OBJECTID=1003", NULL, &rmixppm_interface);
434 if (ret != 0)
435 {
436     RMIX_LOG((RMIX_WARN(unable to export ppm interface)))
437     return -1;
438 }
439
440 PPM_PRINT((
441     PPM_INFO(RMIX initialized)))
442
443 return 0;
444 }
445
446 /*
447 * Finalizes rmix.
448 *
449 * \return 0 on success or -1 on any error
450 */
451 #undef __FUNC__
452 #define __FUNC__ "ppm_fini_rmix"
453 int ppm_fini_rmix()
454 {
455     int ret;
456     int err = 0;
457
458     PPM_PRINT((
459         PPM_INFO(start finalization of RMIX)))
460
461     /* Unexports ppm plug-in */
462     ret = rmix_unexport( &ppm_data.localref_ppm);
463     if (ret != 0)
464     {
465         RMIX_LOG((RMIX_WARN(unable to unexport ppm interface)))
466         err = -1;
467     }
468
469     /* Unexports Harness */
470     ret = harness_rmix_unexport( &ppm_data.localref_harness);
471     if (ret != 0)
472     {
473         RMIX_LOG((RMIX_WARN(unable to unexport Harness kernel)))
474         err = -1;
475     }
476
477     /* Unload RMIX plug-in */
478     ret = harness_plugins_unload( ppm_data.handle_rmix);
479     if (ret != 0)
480     {
481         PPM_PRINT((
482             PPM_WARN(unable to unload RMIX plug-in)))
483         err = -1;
484     }
485
486     PPM_PRINT((
487         PPM_INFO(RMIX finalized)))
488
489     return err;
490 }
```

A. Appendix

```

}
492
494 /*
  * Invokes the Parallel Plug-in Manager execution.
496 *
  * \return 0 on success or -1 on any error
498 */
#define __FUNC__
500 #define __FUNC__ "ppm_start"
int ppm_start()
502 {
    int ret;
504     int mode = 0; /* 1 = replicated, 2 = distributed parallel plug-in */
    char *ppmnode = NULL; /* node of the parallel plug-in manager */
506
    configlist_t *p_nodelist; /* linked list with node address(es) */
508     configlist_t *p_replicatedlist; /* linked list with plug-in name(s) */
    configlist_t *p_distributedlist; /* linked list with plug-in name(s) */
510     configlist_t *p_inputlist; /* linked list with input file(s) */

512     p_nodelist = NULL;
    p_replicatedlist = NULL;
514     p_distributedlist = NULL;
    p_inputlist = NULL;
516
    /* read configuration data */
518     ret = configuration_read_file( &p_nodelist, &p_replicatedlist,
                                   &p_distributedlist, &p_inputlist, &mode,
520                                   &ppmnode);

    if (ret == -1)
522     {
        PPM_PRINT((
524             PPM_WARN(could not read configuration file)))
        configlist_delete_list(&p_nodelist);
526         configlist_delete_list(&p_replicatedlist);
        configlist_delete_list(&p_distributedlist);
528         configlist_delete_list(&p_inputlist);
        FREE(ppmnode);
530         return -1;
    }
532
    /* debug print of the lists */
534     configlist_print_list(p_nodelist);
    configlist_print_list(p_replicatedlist);
536     configlist_print_list(p_distributedlist);
    configlist_print_list(p_inputlist);
538     PPM_PRINT((
        PPM_INFO(mode = %d), mode))
540     PPM_PRINT((
        PPM_INFO(ppmnode = %s), ppmnode))
542
    /******
544     /* PUT YOUR PARALLEL PLUG-IN MANAGER CODE HERE */
    /******
546
    /* load the Harness kernel with the parallel plug-in units */
548     if (mode == 1)
        ret = ppm_load_harnesskernel_replicated( p_nodelist, p_replicatedlist);
550     else if (mode == 2)
        ret = ppm_load_harnesskernel_distributed( &p_nodelist,
552                                                  p_distributedlist,
                                                  ppmnode);

```

A. Appendix

```
554 else
555 {
556     PPM_PRINT((
557         PPM_WARN(no appropriate parallel plug-in type set in conf file)))
558     configlist_delete_list(&p_nodelist);
559     configlist_delete_list(&p_replicatedlist);
560     configlist_delete_list(&p_distributedlist);
561     configlist_delete_list(&p_inputlist);
562     FREE(ppmnode);
563     return -1;
564 }
565 if ( ret == -1 )
566 {
567     PPM_PRINT((
568         PPM_WARN(could not load Harness kernel)))
569     ppm_unload_harnesskernel_soft( p_nodelist);
570     configlist_delete_list(&p_nodelist);
571     configlist_delete_list(&p_replicatedlist);
572     configlist_delete_list(&p_distributedlist);
573     configlist_delete_list(&p_inputlist);
574     FREE(ppmnode);
575     return -1;
576 }
577
578 /******
579 /* PUT YOUR PARALLEL PLUG-IN MANAGER CODE HERE */
580 /******
581
582 /* start the specified plug-in handling for Monte Carlo or image
583    processing */
584 if (mode == 1)
585 {
586     ret = ppm_monte_carlo_loader( p_nodelist, p_replicatedlist,
587                                 p_inputlist);
588     if ( ret == -1 )
589     {
590         PPM_PRINT((
591             PPM_WARN(could not execute parallel plug-in application)))
592     }
593
594     /* after executing the calculation the ppm is responsible for the
595        shutdown of the involved plug-in units */
596     ret = ppm_unload_harnesskernel_soft( p_nodelist);
597     if ( ret == -1 )
598     {
599         PPM_PRINT((
600             PPM_WARN(could not unload Harness kernel)))
601         configlist_delete_list(&p_nodelist);
602         configlist_delete_list(&p_replicatedlist);
603         configlist_delete_list(&p_distributedlist);
604         configlist_delete_list(&p_inputlist);
605         FREE(ppmnode);
606         return -1;
607     }
608 }
609
610 else if (mode == 2)
611 {
612     /* in case of the image processing pipeline, each plug-in shutdown
613        itself when all images were processed */
614     ret = ppm_imageprocessing_loader( p_nodelist, p_distributedlist,
615                                     p_inputlist);
616     if ( ret == -1 )
```

A. Appendix

```
618     {
        PPM_PRINT((
620         PPM_WARN(could not execute parallel plug-in application)))
        /* unload the kernel */
622         ppm_unload_harnesskernel_soft( p_nodelist);
        configlist_delete_list(&p_nodelist);
624         configlist_delete_list(&p_replicatedlist);
        configlist_delete_list(&p_distributedlist);
626         configlist_delete_list(&p_inputlist);
        configlist_delete_list(&ppm_data.usednodes);
628         configlist_delete_list(&ppm_data.freenodes);
        configlist_delete_list(&ppm_data.dist_plugins);
        FREE(ppm_data.sourcedir);
630         FREE(ppm_data.targetdir);
        FREE(ppmnode);
632         return -1;
        }
634     }
    else
636     {
        PPM_PRINT((
638         PPM_WARN(no appropriate parallel plug-in mode selected in
                    configuration file)))
        configlist_delete_list(&p_nodelist);
640         configlist_delete_list(&p_replicatedlist);
        configlist_delete_list(&p_distributedlist);
642         configlist_delete_list(&p_inputlist);
        FREE(ppmnode);
644         return -1;
        }
646     }

648     /******
650     /*
652     /******
        configlist_delete_list(&p_nodelist);
654         configlist_delete_list(&p_replicatedlist);
        configlist_delete_list(&p_distributedlist);
656         configlist_delete_list(&p_inputlist);
        FREE(ppmnode);
658         return 0;
        }
660

662 /*
664 * Loads a replicated parallel plug-in on the Harness kernels specified
        * in the conf file.
        *
666 * \param *nodelist Name of the node(s)
        * \param *pluginlist Name of the parallel plug-in
668 * \return 0 on success or -1 on any error
        */
670 #undef __FUNC__
        #define __FUNC__ "ppm_load_harnesskernel_replicated"
672 int ppm_load_harnesskernel_replicated( configlist_t *nodelist,
                                        configlist_t *pluginlist)
674 {
        int handle;
676         int num_plugins;
        int num_nodes;
678         int i;
        char *command = NULL;
```

A. Appendix

```
680 PPM_PRINT((
681     PPM_INFO(start loading harness kernel and parallel plug-in)))
682
683 num_nodes = configlist_listsize(nodelist);
684 num_plugins = configlist_listsize(pluginlist);
685
686 /* check the availability of nodes and plug-ins */
687 if ((num_plugins == 0) || (num_nodes == 0))
688 {
689     PPM_PRINT((
690         PPM_WARN(no nodes or plug-in names available)))
691     return -1;
692 }
693
694 /* check if a filename was stored */
695 if ( (configlist_check_entries( nodelist) == -1) ||
696     (configlist_check_entries( pluginlist) == -1))
697 {
698     PPM_PRINT((
699         PPM_WARN(some node or plug-in entries have no name)))
700     return -1;
701 }
702
703 /* replicated parallel plug-in */
704 for( i=0; i<num_nodes; i++)
705 {
706 #ifdef DEBUG
707     /* start with debug output */
708     command = (char*)malloc( sizeof(char) * (strlen(nodelist->name) +
709         strlen(pluginlist->name) + 28) );
710     if (command == NULL)
711     {
712         PPM_PRINT((
713             PPM_WARN(could not allocate memory)))
714         return -1;
715     }
716     sprintf( command, "ssh %s harnesssd.debug --load=%s",
717         nodelist->name, pluginlist->name);
718 #else
719     /* start without debug output */
720     command = (char*)malloc( sizeof(char) * (strlen(nodelist->name) +
721         strlen(pluginlist->name) + 22) );
722     if (command == NULL)
723     {
724         PPM_PRINT((
725             PPM_WARN(could not allocate memory)))
726         return -1;
727     }
728     sprintf( command, "ssh %s harnesssd --load=%s", nodelist->name,
729         pluginlist->name);
730 #endif
731     /* execute the ssh command */
732     if ( 0 != harness_processes_execute(&handle, command, NULL) )
733     {
734         PPM_PRINT((
735             PPM_WARN(could not start Harness kernel %s),
736                 nodelist->name))
737     }
738
739     FREE(command);
740     nodelist = nodelist->next;
741 }
742 }
```

A. Appendix

```
744 }
745
746 /*
748 * Loads a distributed parallel plug-in on the Harness kernels specified
749 * in the conf file .
750 *
751 * \param **nodelist Name of the node(s)
752 * \param *pluginlist Name of the parallel plug-in
753 * \param *ppmnode Node running the PPM
754 * \return 0 on success or -1 on any error
755 */
756 #undef __FUNC__
757 #define __FUNC__ "ppm_load_harnesskernel_distributed"
758 int ppm_load_harnesskernel_distributed( configlist_t **nodelist ,
759                                       configlist_t *pluginlist ,
760                                       char *ppmnode)
761 {
762     int handle;
763     int num_plugins;
764     int num_nodes;
765     int ret;
766     char *command = NULL;
767     rmix_remoteref_t *remoteobj;
768     configlist_t *tmpnodelist;
769     configlist_t *tmppluginlist;
770
771     PPM_PRINT((
772         PPM_INFO(start loading harness kernel and distributed plug-in)))
773
774     tmpnodelist = *nodelist;
775     tmppluginlist = pluginlist;
776
777     num_nodes = configlist_listsize(tmpnodelist);
778     num_plugins = configlist_listsize(pluginlist);
779
780     /* check the availability of nodes and plug-ins */
781     if ((num_plugins == 0) || (num_nodes == 0))
782     {
783         PPM_PRINT((
784             PPM_WARN(no nodes or plug-in names available)))
785         return -1;
786     }
787
788     /* check if node and plug-in names were stored */
789     if ( (configlist_check_entries(tmpnodelist) == -1) ||
790         (configlist_check_entries(pluginlist) == -1))
791     {
792         PPM_PRINT((
793             PPM_WARN(some node or plug-in entries have no name)))
794         return -1;
795     }
796
797     /* check if there are not more plug-ins than nodes */
798     if ( num_plugins > num_nodes )
799     {
800         PPM_PRINT((
801             PPM_WARN(more plug-in units than available nodes)))
802         return -1;
803     }
804 }
```


A. Appendix

```
806  /* start loading the plug-ins */
      while(tmppluginlist != NULL)
808  {
      #ifdef DEBUG
810      /* create the ssh start command with debug output */
      command = (char*)malloc( sizeof(char) * (strlen(tmpnodelist->name) +
812      strlen(tmppluginlist->name) + 28) );
      if (command == NULL)
814      {
          PPM_PRINT((
816          PPM_WARN(could not allocate memory)))
          return -1;
818      }
      sprintf( command, "ssh %s harnessd.debug --load=%s",
820      tmpnodelist->name, tmppluginlist->name);
      #else
822      /* create the ssh start command without debug output */
      command = (char*)malloc( sizeof(char) * (strlen(tmpnodelist->name) +
824      strlen(tmppluginlist->name) + 22) );
      if (command == NULL)
826      {
          PPM_PRINT((
828          PPM_WARN(could not allocate memory)))
          return -1;
830      }
      sprintf( command, "ssh %s harnessd --load=%s", tmpnodelist->name,
832      tmppluginlist->name);
      #endif
834      /* execute the ssh command */
      if ( 0 != harness_processes_execute(&handle, command, NULL) )
836      {
          PPM_PRINT((
838          PPM_WARN(could not start Harness kernel %s),
          tmpnodelist->name))
840
          FREE(command);
842
          /* in case of an error delete the current node from the list */
844      ret = configlist_delete_entry( configlist_find_elementposition(
          tmpnodelist->name, *nodelist),
846      nodelist);
848      {
          PPM_PRINT((
850          PPM_WARN(could not delete node from nodelist)))
          return -1;
852      }
      num_nodes--;
854
      /* check if there are still enough nodes */
856      if (num_nodes < num_plugins)
      {
          PPM_PRINT((
858          PPM_WARN(more plug-in units than available nodes)))
          return -1;
860      }
      else
862      {
          /* try the next node */
864      tmpnodelist = tmpnodelist->next;
866      continue;
868      }
      }
```

A. Appendix

```
FREE(command);
870
/* check whether the kernel was loaded */
872
/* set parameters for creating the remote references */
874 command = (char*)malloc( sizeof(char) *
                           (strlen("PROTOCOL=RPC OBJECTID=1002 HOST=%s") +
876                             strlen(tmpnodelist->name) - 1 ));
if (command == NULL)
878 {
    PPM_PRINT((
880         PPM_WARN(could not allocate memory)))
    return -1;
882 }
sprintf( command, "PROTOCOL=RPC OBJECTID=1002 HOST=%s",
884         tmpnodelist->name);

886 /* create references */
ret = rmix_remoteref_create6( &remoteobj, command);
888 if (ret != 0)
{
890     PPM_PRINT((
892         PPM_WARN(could not create remote object references for %s),
                        tmpnodelist->name))

894     FREE(command);

896     /* delete current node from list */
ret = configlist_delete_entry( configlist_find_elementposition(
898         tmpnodelist->name, *nodelist),
                        nodelist);

900     if (ret != 0)
    {
902         PPM_PRINT((
898             PPM_WARN(could not delete node from nodelist)))
        return -1;
904     }

906     num_nodes--;
908     /* check if there are still enough nodes */
if (num_nodes < num_plugins)
910     {
912         PPM_PRINT((
914             PPM_WARN(more plug-in units than available nodes)))
        return -1;
916     }
    else
    {
918         /* try loading on the next node */
        tmpnodelist = tmpnodelist->next;
        continue;
920    }
}

922 FREE(command);

924 /* if the server isn't ready yet and cannot listen to a socket, an error
926     occurs, the sleep should provide a certain amount of time for the
    just loaded plug-in to register the socket */
928 ppm_sleep(2,0);

930 /* contact the plug-in */
ret = rmixmgprocclient_availabilitycheck( remoteobj);
```

A. Appendix

```
932     if (ret != 0)
933     {
934         PPM_PRINT((
935             PPM_WARN(could not contact plug-in on node %s),
936             tmpnodelist->name))
937
938         ret = rmix_remoteref_destroy(&remoteobj);
939         if (ret != 0)
940         {
941             PPM_PRINT((
942                 PPM_WARN(could not destroy remote object references)))
943         }
944
945         /* delete current node from the list */
946         ret = configlist_delete_entry( configlist_find_elementposition(
947             tmpnodelist->name, *nodelist),
948             nodelist);
949
950         if (ret != 0)
951         {
952             PPM_PRINT((
953                 PPM_WARN(could not delete node from nodelist)))
954             return -1;
955         }
956
957         num_nodes--;
958         /* check if there are still enough nodes */
959         if (num_nodes < num_plugins)
960         {
961             PPM_PRINT((
962                 PPM_WARN(more plug-in units than available nodes)))
963             return -1;
964         }
965         else
966         {
967             /* try loading on the next node */
968             tmpnodelist = tmpnodelist->next;
969             continue;
970         }
971     }
972
973     ret = rmix_remoteref_destroy(&remoteobj);
974     if (ret != 0)
975     {
976         PPM_PRINT((
977             PPM_WARN(could not destroy remote object references)))
978     }
979
980     /* load next plug-in on the next free node */
981     tmpnodelist = tmpnodelist->next;
982     tmppluginlist = tmppluginlist->next;
983 }
984
985 /* distinguish between standard loading of distributed plug-ins and repair a
986 pipeline ,
987 if a distributed plug-in is loaded, the lists with used and free nodes
988 have to be generated */
989 if (ppmnode != NULL)
990 {
991     /* copy the nodes to the specific list (usednodes and freenodes) */
992     tmpnodelist = *nodelist;
993     tmppluginlist = pluginlist;
994
995     /* generate used node list */
```

A. Appendix

```
while(tmppluginlist != NULL)
996 {
    configlist_insert_element( tmpnodelist->name, &ppm_data.usednodes);
998     tmpnodelist = tmpnodelist->next;
    tmppluginlist = tmppluginlist->next;
1000 }

/* if there are free nodes, generate free nodes list */
while(tmpnodelist != NULL)
1004 {
    configlist_insert_element( tmpnodelist->name, &ppm_data.freenodes);
1006     tmpnodelist = tmpnodelist->next;
}

/* copy the ppm node name */
1010 ppm_data.ppmnode = (char*)malloc( sizeof(char) * (strlen(ppmnode) + 1));
if (ppm_data.ppmnode == NULL)
1012 {
    PPM_PRINT((
1014         PPM_WARN(could not allocate memory)))
    return -1;
1016 }
strcpy(ppm_data.ppmnode, ppmnode);

/* copy the names of the distributed plug-in units in a global list ,
1020     for a possible pipeline restoration */
tmppluginlist = pluginlist;
1022 while(tmppluginlist != NULL)
{
    ret = configlist_insert_element( tmppluginlist->name,
1024                                     &ppm_data.dist_plugins);
    if (ret != 0)
1026     {
        PPM_PRINT((
1028             PPM_WARN(insert element in the distributed plug-in list)))
        return -1;
1030     }
    tmppluginlist = tmppluginlist->next;
1032 }
}
1034 return 0;
1036 }

1038
/*
1040 * Unoads the Harness kernels on the nodes specified in the conf file.
* The kernels are completely killed inclusive all running plug-ins.
1042 * Careful use of function is advised because of the use of killall.
*
1044 * \param *nodelist Name of the node(s)
* \return 0 on success or -1 on any error
1046 */
#undef __FUNC__
1048 #define __FUNC__ "ppm_unload_harnesskernel"
int ppm_unload_harnesskernel( configlist_t *nodelist)
1050 {
    char *command = NULL;
1052     int i;
    int num_nodes;
1054     int handle;

1056     num_nodes = configlist_listsize(nodelist);
```

A. Appendix

```
1058  /* terminate all kernel */
      for ( i=0; i<num_nodes; i++)
1060  {
1061  #ifdef DEBUG
1062      /* kill harness daemon which used debug */
      command = (char*)malloc( sizeof(char) *
1064                          (strlen("ssh %s killall harnessd.debug") +
                           strlen(nodelist->name) - 1 ));
1066      if (command == NULL)
      {
1068          PPM_PRINT((
              PPM_WARN(could not allocate memory)))
1070          return -1;
      }
1072      sprintf( command, "ssh %s killall harnessd.debug", nodelist->name);
1073  #else
1074      /* kill harness daemon which did not use debug */
      command = (char*)malloc( sizeof(char) *
1076                          (strlen("ssh %s killall harnessd") +
                           strlen(nodelist->name) - 1 ));
1078      if (command == NULL)
      {
1080          PPM_PRINT((
              PPM_WARN(could not allocate memory)))
1082          return -1;
      }
1084      sprintf( command, "ssh %s killall harnessd", nodelist->name);
1085  #endif
1086      /* perform ssh command */
      if ( 0 != harness_processes_execute(&handle, command, NULL) )
1088          PPM_PRINT((
              PPM_WARN(could not shutdown Harness kernel %s), nodelist->name))
1090          FREE(command);
1092      nodelist = nodelist->next;
1093  }
1094  return 0;
1095 }
1096
1097
1098 /*
1099 * Unoads the Harness kernels on the nodes specified in the conf file.
1100 * The kernels are completely killed inclusive all running plug-ins.
1101 * For terminating the kernel the internal shutdown function is used.
1102 *
1103 *
1104 * \param *nodelist  Name of the node(s)
1105 * \return          0 on success or -1 on any error
1106 */
1107 #undef __FUNC__
1108 #define __FUNC__ "ppm_unload_harnesskernel_soft"
1109 int ppm_unload_harnesskernel_soft( configlist_t *nodelist)
1110 {
1111     rmix_remoteref_t *remoteobj;
1112     int i;
1113     int ret;
1114     int num_nodes;
1115
1116     num_nodes = configlist_listsize(nodelist);
1117
1118     /* terminate all kernel */
1119     for ( i=0; i<num_nodes; i++)
1120     {
```

A. Appendix

```
1122     /* prepare remote reference */
1123     ret = ppm_createremoteref( &remoteobj, "1000", nodelist->name);
1124     if (ret != 0)
1125     {
1126         PPM_PRINT((
1127             PPM_WARN(could not create remote object references for %s),
1128             nodelist->name))
1129
1130         nodelist = nodelist->next;
1131         continue;
1132     }
1133
1134     /* call the remote shutdown function */
1135     harnessclient_kernel_shutdown(remoteobj);
1136
1137     ret = rmix_remoteref_destroy(&remoteobj);
1138     if (ret != 0)
1139     {
1140         PPM_PRINT((
1141             PPM_WARN(could not destroy remote object references))
1142
1143
1144         /* move to the next node */
1145         nodelist = nodelist->next;
1146     }
1147
1148     return 0;
1149 }
1150
1151 /*
1152 * The process sleeps a preset time interval.
1153 *
1154 * \param sec_dlay  Delay in seconds
1155 * \param usec_dlay Delay in micro seconds
1156 */
1157 #undef __FUNC__
1158 #define __FUNC__ "ppm_sleep"
1159 void ppm_sleep(int sec_dlay, int usec_dlay)
1160 {
1161     struct timeval tv;
1162     if (sec_dlay > 0)
1163     {
1164         time_t start = time(0);
1165         /* in a loop to be signal-resilient */
1166         for (;;)
1167         {
1168             tv.tv_sec = sec_dlay - (time(0) - start);
1169             if (tv.tv_sec <= 0) break;
1170             tv.tv_usec = 0;
1171             (void)select(0, 0, 0, 0, &tv);
1172         }
1173     }
1174     /* don't worry about signals for usecs */
1175     if (usec_dlay > 0)
1176     {
1177         tv.tv_sec = 0;
1178         tv.tv_usec = usec_dlay;
1179         (void)select(0, 0, 0, 0, &tv);
1180     }
1181 }
1182 }
```

A. Appendix

```
1184
/*
1186 * Creates a remote reference specified by objectid and node. The RPC protocol
* is used.
1188 *
* \param **remoteobj ID of the remote object
1190 * \param *objectid ID of the exported object
* \param *node Name of the node
1192 * \return 0 on success or -1 on any error
*/
1194 #undef __FUNC__
#define __FUNC__ "ppm_createremoteref"
1196 int ppm_createremoteref( rmix_remoteref_t **remoteobj,
                          char *objectid,
1198                          char *node)
{
1200     int ret;
    char *remoteparameters;
1202
    /* prepare parameter for creating remote reference */
1204     remoteparameters = (char*)malloc( sizeof(char) *
                                      (strlen("PROTOCOL=RPC OBJECTID=%s HOST=%s") +
1206                                       strlen(objectid) +
                                       strlen(node) - 3 ));
1208     if (remoteparameters == NULL)
    {
1210         PPM_PRINT((
1212             PPM_WARN(could not allocate memory)))
        return -1;
    }
1214     sprintf( remoteparameters, "PROTOCOL=RPC OBJECTID=%s HOST=%s",
              objectid, node);
1216
    /* create the remote reference */
1218     ret = rmix_remoteref_create6( &(*remoteobj), remoteparameters);
    if (ret != 0)
1220     {
1222         PPM_PRINT((
1224             PPM_WARN(could not create remote object references)))
        return -1;
    }
1226     FREE(remoteparameters);
    return 0;
1228 }

1230
1232 /*
1234 * Monte Carlo section
1236
1238 /*
1240 * Invokes the Monte Carlo parallel plug-in execution.
*
* \param *nodelist Name of the input file
1242 * \param *replicatedlist Name of replicated parallel plug-in
* \param *inputlist List with name(s) of input file(s)
* \return 0 on success or -1 on any error
1244 */
#define __FUNC__
1246 #define __FUNC__ "ppm_monte_carlo_loader"
```

A. Appendix

```
1248 int ppm_monte_carlo_loader( configlist_t *nodelist ,
                             configlist_t *replicatedlist ,
                             configlist_t *inputlist)
1250 {
1252     int ret;
1254     int      num_of_coefs; /* number of coefficients */
1254     unsigned int iterations; /* amount of random numbers */
1256     double *coefs; /* array for storing the coefficients */
1256     double lower_border; /* lower border for the calculation area */
1256     double upper_border; /* upper border of the calculation area */
1258
1260     /* read input data */
1260     ret = ppm_read_integralinput( inputlist , &num_of_coefs , &coefs ,
                                   &iterations , &lower_border , &upper_border);
1262     if ( ret == -1 )
1264     {
1264         PPM_PRINT((
1266             PPM_WARN(could not read input file)))
1266         return -1;
1268     }
1268
1270     /* schedule the integral intervals and call the remote calculation
1270     functions */
1272     ret = ppm_schedule_integral( nodelist , replicatedlist , num_of_coefs ,
                                   coefs , iterations , lower_border ,
                                   upper_border);
1274     if ( ret == -1 )
1276     {
1276         PPM_PRINT((
1278             PPM_WARN(could not schedule the integral data)))
1278         FREE(coefs);
1280         return -1;
1282     }
1282     FREE(coefs);
1284     return 0;
1286 }
1288 /*
1288 * Invokes rmixintegral_integration (client-side method stub).
1290 * Sends the integration parameters to the server and starts the integrstion
1292 * process.
1294 * \param **invokeref The invocation reference (also return).
1294 * \param *remoteobject The remote object reference.
1296 * \param iterations The number of the generated random numbers.
1296 * \param lowerbound The lower boundary of the integration interval.
1298 * \param upperbound The upper boundary of the integration interval.
1298 * \param numcoefs The number of function coefficients.
1300 * \param *coefs The coefficients.
1300 * \return 0 on success or -1 on any error.
1302 */
1302 #undef __FUNC__
1302 #define __FUNC__ "rmixintegralclient_integration_send"
1304 int rmixintegralclient_integration_send( rmix_invokeref_t **invokeref ,
1306                                         rmix_remoteref_t *remoteobject ,
1306                                         unsigned int iterations ,
1306                                         double lowerbound ,
1308                                         double upperbound ,
1308                                         unsigned int numcoefs ,
```


A. Appendix

```
1310                                     double          *coeffs)
1311 {
1312     const void *inary[5];
1313
1314     /* Prepare parameters. */
1315     inary[0] = &iterations;
1316     inary[1] = &lowerbound;
1317     inary[2] = &upperbound;
1318     inary[3] = &numcoeffs;
1319     inary[4] = coeffs;
1320
1321     /* Invoke remote object method by using an asynchronous call. */
1322     if (0 != rmix_send(invokeref, remoteobject, &rmixintegralclient_interface,
1323                     RMIXINTEGRAL_METHODS_INTEGRATION_INDEX, inary, 5))
1324     {
1325         int errno2 = errno;
1326         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
1327         errno = errno2;
1328         return -1;
1329     }
1330
1331     return 0;
1332 }
1333
1334 /*
1335 * Retrieves rmixintegral_integration invocation result (client-side method
1336 * stub).
1337 * Returns the output of a previous asynchronous invocation that sends the
1338 * integration parameters to the server and starts the integration
1339 * process.
1340 *
1341 * \param **invokeref The invocation reference (also return).
1342 * \param *integral The result of the integration.
1343 * \return 0 on success or -1 on any error.
1344 */
1345 #undef __FUNC__
1346 #define __FUNC__ "rmixintegralclient_integration_retrieve"
1347 int rmixintegralclient_integration_retrieve( rmix_invokeref_t **invokeref,
1348                                             double          *integral)
1349 {
1350     int result;
1351     void *outary[2];
1352 #ifdef DEBUG
1353     /* Check invokeref parameter. */
1354     if (NULL == invokeref)
1355     {
1356         errno = EINVAL;
1357         RMIX_LOG((RMIX_WARN(invokeref parameter is null)))
1358         errno = EINVAL;
1359         return -1;
1360     }
1361 #endif /* DEBUG */
1362
1363     /* Prepare output. */
1364     outary[0] = &result;
1365     outary[1] = integral;
1366
1367     /* Retrieve invocation result. */
1368     if (0 != rmix_retrieve(invokeref, outary, 2))
1369     {
1370         int errno2 = errno;
1371         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))

```

A. Appendix

```
        errno = errno2;
1374     return -1;
    }
1376     return result;
}
1378

1380
/*
1382 * Reads the input file , which contains the needed information for integral
* computation.
1384 *
* \param *filename Name of the input file
1386 * \param *number Returns the number of coefficients
* \param **coeffs Returns the array of coefficients
1388 * \param *iterations Returns the amount of random numbers, which are used
* \param *lower Returns the lower boundary of the interval
1390 * \param *upper Returns the upper boundary of the interval
* \return 0 on success or -1 on any error
1392 */
#undef __FUNC__
1394 #define __FUNC__ "ppm_read_integralinput"
int ppm_read_integralinput( configlist_t *inputlist ,
1396     unsigned int *number,
     double **coeffs ,
1398     int *iterations ,
     double *lower,
1400     double *upper)
{
1402     FILE *dat_file; /* input file */
     int i;
1404     int ret;

1406     PPM_PRINT((
         PPM_INFO(start reading input file)))
1408
     /* size of the input file list */
1410     if (configlist_listsize(inputlist) == 0)
     {
1412         PPM_PRINT((
             PPM_WARN(no input file in input list)))
1414         return -1;
     }
1416
     /* only the first mentioned file will be read in */
1418     if (configlist_listsize(inputlist) != 1)
     {
1420         PPM_PRINT((
             PPM_INFO(more than one input file entries found - try
1422                 using first one)))
     }
1424
     /* check if a filename was stored */
1426     if (configlist_check_entries( inputlist) == -1)
     {
1428         PPM_PRINT((
             PPM_WARN(input file entry has no name)))
1430         return -1;
     }
1432
     /* open the input file */
1434     dat_file = fopen(inputlist->name, "r");
     if (dat_file == NULL)
```

A. Appendix

```
1436 {
1437     PPM_PRINT((
1438         PPM_WARN(unable to open input file %s), inputlist->name))
1439     return -1;
1440 }
1441 else
1442 {
1443     PPM_PRINT((
1444         PPM_INFO(input file is opened)))
1445 }
1446
1447 /* read in the number of iterations */
1448 ret = fscanf(dat_file, "%d", iterations);
1449 if (ret == 0 || ret == -1)
1450 {
1451     PPM_PRINT((
1452         PPM_WARN(unable to read iterations from file)))
1453     fclose(dat_file);
1454     return -1;
1455 }
1456
1457 /* read in the boundaries */
1458 ret = fscanf(dat_file, "%lf", lower);
1459 if (ret == 0 || ret == -1)
1460 {
1461     PPM_PRINT((
1462         PPM_WARN(unable to read lower boundary from file)))
1463     fclose(dat_file);
1464     return -1;
1465 }
1466
1467 ret = fscanf(dat_file, "%lf", upper);
1468 if (ret == 0 || ret == -1)
1469 {
1470     PPM_PRINT((
1471         PPM_WARN(unable to read upper boundary from file)))
1472     fclose(dat_file);
1473     return -1;
1474 }
1475
1476 /* read in the number of coefficients */
1477 ret = fscanf(dat_file, "%d", number);
1478 if (ret == 0 || ret == -1)
1479 {
1480     PPM_PRINT((
1481         PPM_WARN(unable to read number of coefficients from
1482             file)))
1483     fclose(dat_file);
1484     return -1;
1485 }
1486
1487 /* create the array */
1488 (*coeffs) = (double *)malloc(sizeof(double) * (*number));
1489 if ((*coeffs) == NULL)
1490 {
1491     PPM_PRINT((
1492         PPM_WARN(unable to allocate memory for coefficients)))
1493     fclose(dat_file);
1494     return -1;
1495 }
1496
1497 /* read in the coefficients */
1498 for (i=0; i<(*number); i++)
```

A. Appendix

```

1500     {
1501         ret = fscanf(dat_file, "%lf", &(*coeffs)[i]);
1502         if (ret == 0 || ret == -1)
1503         {
1504             PPM_PRINT((
1505                 PPM_WARN(unable to read coefficients from file))
1506                 FREE(coeffs);
1507                 fclose(dat_file);
1508                 return -1;
1509         }
1510     }
1511     fclose(dat_file);
1512     return 0;
1513 }
1514
1515 /*
1516  * Partitions the integration interval and invokes the integral plug-in units.
1517  * In case of a failure, a redistribution of missed interval parts is tried.
1518  *
1519  * \param *nodelist      Name of the input file
1520  * \param *pluginlist    Name of the parallel plug-in
1521  * \param numcoeffs      Number of coefficients
1522  * \param *coeffs        Array of coefficients
1523  * \param iterations     Amount of random numbers, which are used
1524  * \param lower_border   Lower boundary of the interval
1525  * \param upper_border   Upper boundary of the interval
1526  * \return               0 on success or -1 on any error
1527  */
1528 #undef __FUNC__
1529 #define __FUNC__ "ppm_schedule_integral"
1530 int ppm_schedule_integral( configlist_t *nodelist,
1531                          configlist_t *pluginlist,
1532                          unsigned int numcoeffs,
1533                          double *coeffs,
1534                          int iterations,
1535                          double lower_border,
1536                          double upper_border)
1537 {
1538     int nodes; /* number of nodes mentioned in conf file */
1539     int workingnodes; /* number of currently working nodes */
1540     double interval; /* interval for one plug-in unit */
1541     double lower; /* lower boundary of temporary calculations */
1542     double upper; /* upper boundary of temporary calculations */
1543     double result = 0; /* result of the integration calculation */
1544     double inter = 0; /* intermediate result of one plug-in */
1545     int count; /* counting variable */
1546     int i, n; /* variables used for loops */
1547     int ret; /* stores return values of functions */
1548     int position; /* variable used to mark positions in arrays */
1549     int offset; /* variable used to mark positions in arrays */
1550     char *remoteparameters; /* used for creating remote reference objects */
1551
1552     unsigned int errors; /* counting occurring errors */
1553     unsigned int *errorpositions; /* array storing the error nodes */
1554     rmix_remoteref_t **integral_tmppremotes; /* temporary remote references */
1555     rmix_remoteref_t **integral_remoterefs; /* references after filtering */
1556     rmix_invokeref_t **integral_invokerefs; /* remote invokation references */
1557
1558     remoteparameters = NULL;
1559     integral_tmppremotes = NULL;
1560     integral_remoterefs = NULL;

```

A. Appendix

```
1562     integral_invokerefs = NULL;
1564     PPM_PRINT((
1565         PPM_INFO(start scheduling)))
1566
1567     /* size of the node list */
1568     if ( (nodes = configlist_listsize(nodelist)) == 0)
1569     {
1570         PPM_PRINT((
1571             PPM_WARN(no nodes available)))
1572         return -1;
1573     }
1574
1575     /* check if node names were stored */
1576     if (configlist_check_entries( nodelist) == -1)
1577     {
1578         PPM_PRINT((
1579             PPM_WARN(one or more node entries have no node name)))
1580         return -1;
1581     }
1582
1583     /* size of plugin name list */
1584     if ( configlist_listsize(pluginlist) == 0)
1585     {
1586         PPM_PRINT((
1587             PPM_WARN(no plugin available)))
1588         return -1;
1589     }
1590
1591     /* only the first mentioned plug-in will be used */
1592     if (configlist_listsize(pluginlist) != 1)
1593     {
1594         PPM_PRINT((
1595             PPM_INFO(more than one plug-in entry found - try using first one)))
1596     }
1597
1598     /* check if a plugin name was stored */
1599     if (configlist_check_entries( pluginlist) == -1)
1600     {
1601         PPM_PRINT((
1602             PPM_WARN(plugin entry has no stored name)))
1603         return -1;
1604     }
1605
1606     /* allocate memory for the temporary remote references */
1607     integral_tmppremotes = (rmix_remoteref_t**) malloc(sizeof(rmix_remoteref_t*)
1608                                                         * nodes);
1609
1610     if (integral_tmppremotes == NULL)
1611     {
1612         PPM_PRINT((
1613             PPM_WARN(could not allocate memory for remote references)))
1614         return -1;
1615     }
1616
1617     /* workingnodes will store the number of currently available nodes */
1618     workingnodes = nodes;
1619
1620     /* create remote references for all nodes mentioned in the conf file */
1621     for ( i=0; i<nodes; i++)
1622     {
1623         /* set parameters for creating the remote references */
1624         remoteparameters = (char*)malloc( sizeof(char) *
1625                                           (strlen("PROTOCOL=RPC OBJECTID=1001 HOST=%s") +
```

A. Appendix

```

1626         strlen(nodelist->name) - 1 ));
1627     if (remoteparameters == NULL)
1628     {
1629         PPM_PRINT((
1630             PPM_WARN(could not allocate memory for remote parameters)))
1631
1632         /* destroy already created remote references */
1633         for ( n=0; n<nodes; n++)
1634         {
1635             ret = rmix_remoteref_destroy( &integral_tmpremotes[i]);
1636             if (ret != 0)
1637             {
1638                 PPM_PRINT((
1639                     PPM_WARN(could not destroy remote object references)))
1640             }
1641
1642             /* free allocated memory */
1643             FREE(integral_tmpremotes);
1644
1645             return -1;
1646         }
1647         sprintf( remoteparameters, "PROTOCOL=RPC OBJECTID=1001 HOST=%s",
1648             nodelist->name);
1649
1650         /* create references */
1651         ret = rmix_remoteref_create6( &integral_tmpremotes[i],
1652             remoteparameters);
1653         if (ret != 0)
1654         {
1655             PPM_PRINT((
1656                 PPM_WARN(could not create remote object references)))
1657
1658             /* if the remote reference cannot be created decrease the number
1659              * of currently available nodes */
1660             workingnodes--;
1661             /* tag the node as not working */
1662             integral_tmpremotes[i] = NULL;
1663         }
1664
1665         FREE(remoteparameters);
1666
1667         /* create remote reference for the next node */
1668         nodelist = nodelist->next;
1669     }
1670
1671     /* allocate memory for the remote references */
1672     integral_remoterefs = (rmix_remoteref_t**) malloc(sizeof(rmix_remoteref_t*)
1673         * workingnodes);
1674     if (integral_remoterefs == NULL)
1675     {
1676         PPM_PRINT((
1677             PPM_WARN(could not allocate memory for remote references)))
1678
1679         /* destroy already created temporary remote references */
1680         for ( i=0; i<nodes; i++)
1681         {
1682             ret = rmix_remoteref_destroy( &integral_tmpremotes[i]);
1683             if (ret != 0)
1684             {
1685                 PPM_PRINT((
1686                     PPM_WARN(could not destroy remote object references)))
1687             }
1688         }
1689     }

```

A. Appendix

```
1688     }
1689     FREE(integral_tmppremotes);
1690     return -1;
1691 }
1692
1693 /* allocate memory for the invoke references */
1694 integral_invokerefs = (rmix_invokeref_t**) malloc(sizeof(rmix_invokeref_t*)
1695                                                * workingnodes);
1696 if (integral_invokerefs == NULL)
1697 {
1698     PPM_PRINT((
1699         PPM_WARN(could not allocate memory for invoke references)))
1700
1701     /* destroy already created temporary remote references */
1702     for ( i=0; i<nodes; i++)
1703     {
1704         ret = rmix_remoteref_destroy( &integral_tmppremotes[i]);
1705         if (ret != 0)
1706         {
1707             PPM_PRINT((
1708                 PPM_WARN(could not destroy remote object references)))
1709         }
1710     }
1711     FREE(integral_tmppremotes);
1712     FREE(integral_remoterefs);
1713     return -1;
1714 }
1715
1716 /* identifies the current position in the remote references array for
1717    storing the next currently available node reference */
1718 count = 0;
1719
1720 /* copy the working references */
1721 for ( i=0; i<nodes; i++)
1722 {
1723     if (integral_tmppremotes[i] != NULL)
1724     {
1725         /* copy reference if it isn't NULL */
1726         ret = rmix_remoteref_duplicate( &integral_remoterefs[count],
1727                                       integral_tmppremotes[i]);
1728         if (ret != 0)
1729         {
1730             PPM_PRINT((
1731                 PPM_WARN(could not copy remote object references)))
1732
1733             /* destroy already created temporary remote references */
1734             for ( n=0; n<nodes; n++)
1735             {
1736                 ret = rmix_remoteref_destroy( &integral_tmppremotes[n]);
1737                 if (ret != 0)
1738                 {
1739                     PPM_PRINT((
1740                         PPM_WARN(could not destroy remote object
1741                             references)))
1742                 }
1743             }
1744
1745             /* destroy already copied working remote references */
1746             for ( n=0; n<workingnodes; n++)
1747             {
1748                 ret = rmix_remoteref_destroy( &integral_remoterefs[n]);
1749                 if (ret != 0)
```

A. Appendix

```
1752         {
1753             PPM_PRINT((
1754                 PPM_WARN(could not destroy remote object
1755                     references)))
1756         }
1757         FREE(integral_tmppremotes);
1758         FREE(integral_remoterefs);
1759         FREE(integral_invokerefs);
1760         return -1;
1761     }
1762     /* destroy copied reference */
1763     ret = rmix_remoteref_destroy( &integral_tmppremotes[i]);
1764     if (ret != 0)
1765     {
1766         PPM_PRINT((
1767             PPM_WARN(could not destroy remote object references)))
1768     }
1769     count++;
1770 }
1771 }
1772 FREE(integral_tmppremotes);
1773
1774
1775 /* if the server isn't ready yet and cannot listen to a socket, an error
1776 occurs, the sleep should provide a certain amount of time for the just
1777 loaded plug-in to register the socket */
1778 ppm_sleep(2,0);
1779
1780 /* prepare the calculation data for distribution */
1781
1782 /* interval size of one plug-in unit */
1783 interval = (upper_border - lower_border) / workingnodes;
1784
1785 /* lower boundary for the first plug-in unit */
1786 lower = lower_border;
1787
1788 /* upper boundary for the first plug-in unit */
1789 upper = lower + interval;
1790
1791 /* counting occuring errors */
1792 errors = 0;
1793
1794 /* call remote functions on the working nodes */
1795 for ( i=0; i<workingnodes; i++)
1796 {
1797     /* send data to the plug-ins asynchronously */
1798     if (integral_remoterefs[i] != NULL)
1799     {
1800         ret = rmixintegralclient_integration_send( &integral_invokerefs[i],
1801             integral_remoterefs[i],
1802             iterations,
1803             lower,
1804             upper,
1805             numcoeffs,
1806             coeffs);
1807
1808         if (ret != 0)
1809         {
1810             /* if sending failed increase error counter */
1811             errors++;
1812             /* and tag the node as not available anymore */
1813             rmix_remoteref_destroy( &integral_remoterefs[i]);

```


A. Appendix

```
1814         /* by setting the reference NULL */
1815         integral_remoterefs[i] = NULL;
1816     }
1817
1818     /* borders for the next plug-in unit */
1819     lower = upper;
1820     upper = lower + interval;
1821 }
1822 else
1823     integral_invokerefs[i] = NULL;
1824 }
1825
1826 /* call remote functions for receiving the results */
1827 for ( i = 0; i < workingnodes; i++)
1828 {
1829     /* only ask working nodes for results */
1830     if (integral_remoterefs[i] != NULL)
1831     {
1832         /* retrieve the result */
1833         ret = rmixintegralclient_integration_retrieve(
1834             &integral_invokerefs[i], &inter) ;
1835         if (ret != 0)
1836         {
1837             /* if retrieving fails increase error counter */
1838             errors++;
1839
1840             /* and tag the node as not available anymore */
1841             rmix_remoteref_destroy( &integral_remoterefs[i]);
1842             /* by setting the reference NULL */
1843             integral_remoterefs[i] = NULL;
1844         }
1845         else
1846         {
1847             /* otherwise sum the result */
1848             result += inter;
1849         }
1850     }
1851 }
1852
1853 /* if all nodes failed, there is no redistribution of work possible */
1854 if (errors == workingnodes)
1855 {
1856     PPM_PRINT((
1857         PPM_WARN(terminating program - all communication tries failed)))
1858
1859     /* destroy remote references */
1860     for ( i=0; i<workingnodes; i++)
1861     {
1862         ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
1863         if (ret != 0)
1864         {
1865             PPM_PRINT((
1866                 PPM_WARN(could not destroy remote object references)))
1867         }
1868     }
1869     FREE(integral_remoterefs);
1870     FREE(integral_invokerefs);
1871     return -1;
1872 }
1873
1874 count = 0;
1875 /* if errors occurred start the redistribution of the failed calculations */
```

A. Appendix

```
1878 if (errors != 0)
1879 {
1880     PPM_PRINT((
1881         PPM_INFO(starting error recovery)))
1882
1883     /* the failed nodes will be stored in an array and with the help of
1884        the node number the missing calculation part will be recovered */
1885     errorpositions = (unsigned int*)malloc( sizeof(unsigned int) * errors);
1886     if (errorpositions == NULL)
1887     {
1888         PPM_PRINT((
1889             PPM_WARN(could not allocate memory for error positions)))
1890
1891         /* destroy remote references */
1892         for ( i=0; i<workingnodes; i++)
1893         {
1894             ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
1895             if (ret != 0)
1896             {
1897                 PPM_PRINT((
1898                     PPM_WARN(could not destroy remote object references)))
1899             }
1900         }
1901         FREE(integral_remoterefs);
1902         FREE(integral_invokerefs);
1903         return -1;
1904     }
1905
1906     /* search for the failed nodes and store the rank number, the position
1907        in the nodelist identifies the chunk of data each node received */
1908     for ( i=0; i<workingnodes; i++)
1909     {
1910         if ( integral_remoterefs[i] == NULL)
1911         {
1912             errorpositions[count] = i;
1913         }
1914     }
1915
1916     /* nodes are the currently available nodes, if nodes fail during the
1917        redistribution process nodes is decreased */
1918     nodes = workingnodes;
1919
1920     /* count determines the failed calculation parts, if a calculation part
1921        is solved by redistribution count is decreased */
1922     count = errors;
1923
1924     /* while still calculation parts are missing redistribute */
1925     while(count != 0)
1926     {
1927         /* if all nodes failed the redistribution isn't possible anymore */
1928         if (nodes == 0)
1929         {
1930             PPM_PRINT((
1931                 PPM_WARN(all nodes failed during redistribution process)))
1932
1933             /* destroy remote references */
1934             for ( i=0; i<workingnodes; i++)
1935             {
1936                 ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
1937                 if (ret != 0)
1938                 {
1939                     PPM_PRINT((
1940                         PPM_WARN(could not destroy remote object
```

A. Appendix

```
1940             references)))
1941         }
1942     }
1943     FREE(integral_remoterefs);
1944     FREE(integral_invokerefs);
1945     FREE(errorpositions);
1946     return -1;
1947 }
1948
1949 /* for loadbalancing position stores the node which got a new chunk
1950    for calculation as the last one */
1951 position = 0;
1952
1953 /* solve all missed parts of the failed nodes */
1954 for ( i=0; i<errors; i++)
1955 {
1956     /* -1 tags the failed chunk as calculated */
1957     if (errorpositions[i] != -1)
1958     {
1959         /* calculate the lower boundary of the missed chunk */
1960         lower = lower_border + errorpositions[i] * (upper_border -
1961             lower_border) / workingnodes;
1962
1963         /* calculate the upper boundary of the missed chunk */
1964         upper = lower + (upper_border - lower_border)
1965             / workingnodes;
1966
1967         /* send the chunk to the next free and available node */
1968         for ( n=position; n<workingnodes; n++)
1969         {
1970             /* find an available node */
1971             if(integral_remoterefs[n] != NULL)
1972             {
1973                 /* send dataset */
1974                 ret = rmixintegralclient_integration_send(
1975                     &integral_invokerefs[n],
1976                     integral_remoterefs[n],
1977                     iterations,
1978                     lower,
1979                     upper,
1980                     numcoeffs,
1981                     coeffs);
1982
1983                 if (ret != 0)
1984                 {
1985                     /* if the sending fails tag the node as not */
1986                     ret = rmix_remoteref_destroy(
1987                         &integral_remoterefs[n]);
1988
1989                     if (ret != 0)
1990                     {
1991                         PPM_PRINT((
1992                             PPM_WARN(could not destroy remote object
1993                                 references)))
1994                     }
1995                     /* available anymore */
1996                     integral_remoterefs[n] = NULL;
1997
1998                     /* decrease the number of currently available
1999                        nodes */
2000                     nodes--;
2001
2002                     /* try to send the data chunk to the next
2003                        available node */
2004                     continue;

```

A. Appendix

```
2004         }
2005         else
2006         {
2007             /* store the position of the node which got the
2008              chunk */
2009             position = n;
2010
2011             /* calculate the next missing chunk and
2012              redistribute it */
2013             break;
2014         }
2015     }
2016 }
2017 }
2018
2019 position = 0;
2020 offset = 0;
2021
2022 /* collect the results */
2023 for ( i=0; i<workingnodes; i++)
2024 {
2025     /* from the still reachable nodes */
2026     if (integral_invokerefs[i] != NULL)
2027     {
2028         /* retrieve result */
2029         ret = rmixintegralclient_integration_retrieve(
2030             &integral_invokerefs[i], &inter) ;
2031         if (ret != 0)
2032         {
2033             /* if the retrieving failed tag the node as failed */
2034             ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
2035             if (ret != 0)
2036             {
2037                 PPM_PRINT((
2038                     PPM_WARN(could not destroy remote object
2039                             references)))
2040             }
2041             integral_remoterefs[i] = NULL;
2042
2043             /* decrease the number of available nodes */
2044             nodes--;
2045
2046             /* the algorithm assumes that all requests come back
2047              in the order as they were sent (depending on the
2048              position of their rank in the error array)
2049              if a part was not solved, the current position in
2050              the array of failed nodes cannot be set as solved, so
2051              an offset is increased */
2052             offset++;
2053         }
2054     }
2055     else
2056     {
2057         /* sum up the result */
2058         result += inter;
2059
2060         /* one missing calculation part is solved */
2061         count--;
2062
2063         /* the current position in the array of failed nodes
2064          will be set as solved */
2065         for ( n=0; n<errors; n++)
2066         {
```

A. Appendix

```
2066     /* find the next unsolved chunk */
2067     if(errorpositions[n] != -1)
2068     {
2069         /* if a chunk could not be received, the offset
2070            was set, the failed chunks will be overjumped
2071            and the next correct received chunk is reset
2072            */
2073         if (offset != 0)
2074         {
2075             /* store the offset */
2076             if (position == 0)
2077                 position = offset;
2078             /* decrease offset and */
2079             offset--;
2080             /* find the next unsolved chunk */
2081             continue;
2082         }
2083         else
2084         {
2085             /* tag the chunk as solved */
2086             errorpositions[n] = -1;
2087             /* restore the offset */
2088             offset = position;
2089             position = 0;
2090
2091             /* leave loop to get next result */
2092             break;
2093         }
2094     }
2095 }
2096 }
2097 }
2098 }
2100     FREE(errorpositions);
2101 }
2102
2103 PPM_PRINT((
2104     PPM_INFO(++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++)))
2105 PPM_PRINT((
2106     PPM_INFO(integral = %lf), result))
2107 PPM_PRINT((
2108     PPM_INFO(++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++)))
2109
2110 /* destroy remote references */
2111 for ( i=0; i<workingnodes; i++)
2112 {
2113     ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
2114     if (ret != 0)
2115     {
2116         PPM_PRINT((
2117             PPM_WARN(could not destroy remote object references)))
2118     }
2119 }
2120
2121 FREE(integral_remoterefs);
2122 FREE(integral_invokerefs);
2123 return 0;
2124 }
2125
2126
2127
2128 *
```

A. Appendix

```

2130 * Image Processing section
2131 *
2132 *****/
2133 /*
2134 * Invokes the image processing pipeline.
2135 *
2136 * \param *nodelist      Name of the node(s)
2137 * \param *distributedlist  Name of replicated parallel plug-in
2138 * \param *inputlist      List with name(s) of input file(s)
2139 * \return                0 on success or -1 on any error
2140 */
2141 #undef __FUNC__
2142 #define __FUNC__ "ppm_imageprocessing_loader"
2143 int ppm_imageprocessing_loader( configlist_t *nodelist,
2144                               configlist_t *distributedlist,
2145                               configlist_t *inputlist)
2146 {
2147     int ret;
2148     int i;
2149     unsigned int nodes;
2150     unsigned int plugins;
2151
2152     rmix_remoteref_t **imgproc_remoterefs = NULL;
2153
2154     char *p_sourcedir;      /* src directory with image files */
2155     char *p_targetdir;      /* target directory for storing img */
2156
2157     configlist_t *tmp_pointer;
2158
2159     p_sourcedir = NULL;
2160     p_targetdir = NULL;
2161     tmp_pointer = NULL;
2162
2163     /* read input data */
2164     ret = ppm_read_imageinput( inputlist, &p_sourcedir, &p_targetdir);
2165     if ( ret == -1 )
2166     {
2167         PPM_PRINT((
2168             PPM_WARN(could not read input file)))
2169         FREE(p_sourcedir);
2170         FREE(p_targetdir);
2171         return -1;
2172     }
2173
2174     /* print debug output */
2175     PPM_PRINT((
2176         PPM_INFO(source dir %s), p_sourcedir))
2177     PPM_PRINT((
2178         PPM_INFO(target dir %s), p_targetdir))
2179
2180     /* copy the source and the target directories into global variable, for a
2181     possible pipeline restoration */
2182     ppm_data.sourcedir = (char*)malloc(sizeof(char) * (strlen(p_sourcedir) + 1));
2183     if (ppm_data.sourcedir == NULL)
2184     {
2185         PPM_PRINT((
2186             PPM_WARN(could not allocate memory input file)))
2187         FREE(p_sourcedir);
2188         FREE(p_targetdir);
2189         return -1;
2190     }
2191     strcpy( ppm_data.sourcedir, p_sourcedir);

```

A. Appendix

```
2192 ppm_data.targetdir = (char*)malloc(sizeof(char) * (strlen(p_targetdir) + 1));
2193 if (ppm_data.targetdir == NULL)
2194 {
2195     PPM_PRINT((
2196         PPM_WARN(could not allocate memory input file)))
2197     FREE(p_sourcedir);
2198     FREE(p_targetdir);
2199     return -1;
2200 }
2201 strcpy( ppm_data.targetdir , p_targetdir);
2202
2203 /* size of the node list */
2204 if ( (nodes = configlist_listsize(nodelist)) == 0)
2205 {
2206     PPM_PRINT((
2207         PPM_WARN(no nodes available)))
2208     return -1;
2209 }
2210
2211 /* check if node names were stored */
2212 if (configlist_check_entries( nodelist) == -1)
2213 {
2214     PPM_PRINT((
2215         PPM_WARN(node entry has no node name)))
2216     return -1;
2217 }
2218
2219 /* size of plugin name list */
2220 if ( (plugins = configlist_listsize(distributedlist)) == 0)
2221 {
2222     PPM_PRINT((
2223         PPM_WARN(no plugin available)))
2224     return -1;
2225 }
2226
2227 /* allocate memory for the remote references */
2228 imgproc_remoterefs = (rmix_remoteref_t**) malloc(sizeof(rmix_remoteref_t*)
2229 * plugins);
2230
2231 if (imgproc_remoterefs == NULL)
2232 {
2233     PPM_PRINT((
2234         PPM_WARN(could not allocate memory for remote references)))
2235     return -1;
2236 }
2237
2238 /* create remote references for all used nodes */
2239 tmp_pointer = nodelist;
2240 for ( i=0; i<plugins; i++)
2241 {
2242     ret = ppm_createremoteref( &imgproc_remoterefs[i], "1002",
2243         tmp_pointer->name);
2244     if (ret != 0)
2245     {
2246         PPM_PRINT((
2247             PPM_WARN(could not create remote object)))
2248         return -1;
2249     }
2250     tmp_pointer = tmp_pointer->next;
2251 }
2252
2253 /* if the server isn't ready yet and cannot listen to a socket, an error
2254 occurs, the sleep should provide a certain amount of time for the just
loaded plug-in to register the socket */
```

A. Appendix

```
ppm_sleep(2,0);
2256

tmp_pointer = nodelist;
2258

/* initialize the distributed parallel plug-in */
/* "nada" identifies variables, which are not necessary for a particular
2260 pipeline unit, i.e. if a plug-in has as the name for the predecessor
2262 "nada", it knows that it has no predecessor */
for ( i=0; i<plugins; i++)
2264 {
    /* the first plug-in unit has to open the images */
    if(i==0)
2266 {
        /* check if the "pipeline" only consists of one plug-in */
2268 if (plugins != 1)
            ret = rmixmapprocclient_initpipeline( imgproc_remoterefs[i], i,
2270 p_sourcedir, "nada",
            tmp_pointer->next->name,
2272 "nada", ppm_data.ppmnode);
        else
2274 ret = rmixmapprocclient_initpipeline( imgproc_remoterefs[i], i,
            p_sourcedir, p_targetdir,
2276 "nada", "nada",
            ppm_data.ppmnode);
2278
        if (ret != 0)
2280 {
            PPM_PRINT((
2282 PPM_WARN(could not initialize pipeline)))
2284
            /* destroy remote and invoke references */
2286 for ( i=0; i<plugins; i++)
            {
                ret = rmixmapremoteref_destroy( &imgproc_remoterefs[i]);
2288 if (ret != 0)
                {
                    PPM_PRINT((
2290 PPM_WARN(could not destroy remote object references)))
2292                }
            }
2294 FREE(imgproc_remoterefs);
2296 FREE(p_sourcedir);
2298 FREE(p_targetdir);
2300 return -1;
        }
    }
2302
    /* the last plug-in unit has to store the images */
2304 else if(i==plugins-1)
    {
        ret = rmixmapprocclient_initpipeline( imgproc_remoterefs[i], i,
2306 "nada", p_targetdir, "nada",
            tmp_pointer->prev->name,
2308 ppm_data.ppmnode);

        if (ret != 0)
2310 {
            PPM_PRINT((
2312 PPM_WARN(could not initialize pipeline)))
2314
            /* destroy remote and invoke references */
2316 for ( i=0; i<plugins; i++)
            {
```


A. Appendix

```
2318         ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
2320         if (ret != 0)
2322             {
2324                 PPM_PRINT((
2326                     PPM_WARN(could not destroy remote object references)))
2328             }
2330         FREE(imgproc_remoterefs);
2332         FREE(p_sourcedir);
2334         FREE(p_targetdir);
2336         return -1;
2338     }
2340 /* every plug-in unit in between has to forward the images and
2342 acknowledgments */
2344 else
2346 {
2348     ret = rmiximgprocclient_initpipeline( imgproc_remoterefs[i], i,
2350                                         "nada", "nada",
2352                                         tmp_pointer->next->name,
2354                                         tmp_pointer->prev->name,
2356                                         ppm_data.ppmnode);
2358     if (ret != 0)
2360     {
2362         PPM_PRINT((
2364             PPM_WARN(could not initialize pipeline)))
2366
2368         /* destroy remote and invoke references */
2370         for ( i=0; i<plugins; i++)
2372         {
2374             ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
2376             if (ret != 0)
2378             {
2380                 PPM_PRINT((
2382                     PPM_WARN(could not destroy remote object references)))
2384             }
2386             FREE(imgproc_remoterefs);
2388             FREE(p_sourcedir);
2390             FREE(p_targetdir);
2392             return -1;
2394         }
2396     }
2398     tmp_pointer = tmp_pointer->next;
2400 }
2402 /* invoke the first plug-in of the pipeline */
2404 ret = rmiximgprocclient_invokepipeline_oneway (imgproc_remoterefs[0]);
2406 if (ret != 0)
2408 {
2410     PPM_PRINT((
2412         PPM_WARN(could not invoke the pipeline)))
2414
2416     /* destroy remote and invoke references */
2418     for ( i=0; i<plugins; i++)
2420     {
2422         ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
2424         if (ret != 0)
2426         {
```

A. Appendix

```
2382         PPM_PRINT((
                PPM_WARN(could not destroy remote object references)))
2384     }
    FREE(imgproc_remoterefs);
2386
    FREE(p_sourcedir);
2388     FREE(p_targetdir);
2390     return -1;
    }
2392
    /* destroy remote and invoke references */
2394     for ( i=0; i<plugins; i++)
    {
2396         ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
        if (ret != 0)
2398         {
            PPM_PRINT((
2400                 PPM_WARN(could not destroy remote object references)))
        }
2402     }
    FREE(imgproc_remoterefs);
2404
    FREE(p_sourcedir);
2406     FREE(p_targetdir);
2408     return 0;
    }
2410
2412 /*
    * Reads the input file , which contains the needed information for image
2414 * processing.
    *
2416 * \param *inputlist    List with name(s) of input file(s)
    * \param **sourcedir  Returns list with directories of images
2418 * \param **targetdir  Returns target directory
    * \return             0 on success or -1 on any error
2420 */
    #undef __FUNC__
2422 #define __FUNC__ "ppm_read_imageinput"
    int ppm_read_imageinput( configlist_t *inputlist ,
2424                          char **sourcedir ,
                          char **targetdir)
2426 {
    /* build the input file structure */
2428     cfg_opt_t opts[] =
    {
2430         CFG_STR( SOURCEDIR, NULL, CFGF_NONE),
        CFG_STR( TARGETDIR, NULL, CFGF_NONE),
2432         CFG_END()
    };
2434
    cfg_t *cfg;
2436
    PPM_PRINT((
2438         PPM_INFO(start reading imgproc input file)))
2440
    /* size of the input file list */
    if (configlist_listsize(inputlist) == 0)
2442     {
        PPM_PRINT((
```

A. Appendix

```
2444     PPM_WARN(no input file in input list))
2445     return -1;
2446 }
2447
2448 /* only the second mentioned file will be read in */
2449 if (configlist_listsize(inputlist) != 1)
2450 {
2451     PPM_PRINT((
2452         PPM_INFO(more than one input file entries found - try
2453                 using second one)))
2454 }
2455
2456 /* check if a filename was stored */
2457 if (configlist_check_entries( inputlist) == -1)
2458 {
2459     PPM_PRINT((
2460         PPM_WARN(input file entry has no name)))
2461     return -1;
2462 }
2463
2464 /* initialize the conf file structure */
2465 cfg = cfg_init(opts, CFGF_NONE);
2466
2467 /* read the input file */
2468 if (cfg_parse(cfg, inputlist->next->name) == CFG_PARSE_ERROR)
2469     return -1;
2470
2471 /* extract target directory */
2472 if (cfg_getstr(cfg, TARGETDIR) == NULL)
2473 {
2474     (*targetdir) = NULL;
2475     PPM_PRINT((
2476         PPM_WARN(no target directory entry in input file)))
2477     return -1;
2478 }
2479
2480 /* allocate memory for the target directory */
2481 (*targetdir) = (char*)malloc(sizeof(char)*(strlen(cfg_getstr(cfg,
2482                                     TARGETDIR))+1));
2483 if ( (*targetdir) == NULL )
2484 {
2485     PPM_PRINT((
2486         PPM_WARN(allocating of memory failed)))
2487     return -1;
2488 }
2489
2490 /* store the information */
2491 strcpy( (*targetdir), cfg_getstr(cfg, TARGETDIR));
2492
2493 /* extract source directory */
2494 if (cfg_getstr( cfg, SOURCEDIR) == NULL)
2495 {
2496     (*sourcedir) = NULL;
2497     PPM_PRINT((
2498         PPM_WARN(no source directory entry in input file)))
2499     return -1;
2500 }
2501 else
2502 {
2503     (*sourcedir) = (char*)malloc( sizeof(char) *
2504                                   (strlen(cfg_getstr(cfg, SOURCEDIR))+1));
2505     if ( (*sourcedir) == NULL )
```

A. Appendix

```
2508     {
2509         PPM_PRINT((
2510             PPM_WARN(allocating of memory failed)))
2511         return -1;
2512     }
2513
2514     /* store the information */
2515     strcpy( (*sourcedir), cfg_getstr(cfg, SOURCEDIR));
2516 }
2517
2518 /* delete the conf file structure */
2519 cfg_free(cfg);
2520 return 0;
2521 }
2522
2523 /*
2524 * Initialises the pipeline (client-side method stub).
2525 *
2526 * \param *remoteobj ID of the remote object
2527 * \param filter Image filter which will be used
2528 * \param *sourcedir Directory containing image sources.
2529 * \param *targetdir Directory containing processed images
2530 * \param *successor ID of the next plug-in in the pipeline
2531 * \param *ppmnode Node with the running PPM
2532 * \return 0 on success or -1 on any error
2533 */
2534 #undef __FUNC__
2535 #define __FUNC__ "rmiximgprocclient_initpipeline"
2536 int rmiximgprocclient_initpipeline ( rmix_remoteref_t *remoteobj,
2537                                     unsigned int filter,
2538                                     char *sourcedir,
2539                                     char *targetdir,
2540                                     char *successor,
2541                                     char *predecessor,
2542                                     char *ppmnode)
2543 {
2544     const void *inary[6];
2545     void *outary[1];
2546     int result;
2547
2548     /* Prepare parameters. */
2549     inary[0] = &filter;
2550     inary[1] = sourcedir;
2551     inary[2] = targetdir;
2552     inary[3] = successor;
2553     inary[4] = predecessor;
2554     inary[5] = ppmnode;
2555
2556     outary[0] = &result;
2557
2558     /* Invoke remote object method. */
2559     if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
2560                         RMIXIMGPROC_METHODS_INITPIPELINE_INDEX, outary, 1,
2561                         inary, 6))
2562     {
2563         int errno2 = errno;
2564         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
2565         errno = errno2;
2566         return -1;
2567     }
2568
2569     return result;

```

A. Appendix

```
2570 }

2572
2574 /*
2575  * Invokes the pipeline (client-side method stub).
2576  *
2577  * \param  *remoteobj  ID of the remote object
2578  * \return          0 on success or -1 on any error
2579  */
2580 #undef __FUNC__
2581 #define __FUNC__ "rmiximgprocclient_invokepipeline_oneway"
2582 int rmiximgprocclient_invokepipeline_oneway (rmix_remoteref_t *remoteobj)
2583 {
2584     /* Invoke remote object method. */
2585     if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface,
2586                          RMXIMGPROC_METHODS_INVOKEPIPELINE_INDEX, NULL, 0))
2587     {
2588         int errno2 = errno;
2589         RMX_LOG((RMIX_WARN(unable to invoke remote object method)))
2590         errno = errno2;
2591         return -1;
2592     }
2593     return 0;
2594 }

2596
2598 /*
2599  * Checks the availability of a plug-in (client-side method stub).
2600  *
2601  * \param  *remoteobj  ID of the remote object
2602  * \return          0 on success or -1 on any error
2603  */
2604 #undef __FUNC__
2605 #define __FUNC__ "rmiximgprocclient_availabilitycheck"
2606 int rmiximgprocclient_availabilitycheck( rmix_remoteref_t *remoteobj)
2607 {
2608     void      *outary[1];
2609     int       result;
2610
2611     outary[0] = &result;
2612
2613     /* Invoke remote object method. */
2614     if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
2615                          RMXIMGPROC_METHODS_AVAILABILITYCHECK_INDEX, outary, 1,
2616                          NULL, 0))
2617     {
2618         int errno2 = errno;
2619         RMX_LOG((RMIX_WARN(unable to invoke remote object method)))
2620         errno = errno2;
2621         return -1;
2622     }
2623     return 0;
2624 }

2626
2628 /*
2629  * Update the image counter of the plug-in (client-side method stub).
2630  *
2631  * \param  *remoteobj  ID of the remote object
2632  * \param  imagecounter  New image counter value
2633  * \return          0 on success or -1 on any error

```

A. Appendix

```
2634 */
2634 #undef __FUNC__
2634 #define __FUNC__ "rmiximgprocclient_updateimagecounter"
2636 int rmiximgprocclient_updateimagecounter( rmix_remoteref_t *remoteobj,
2636                                         unsigned int    imagecounter)
2638 {
2640     const void *inary[1];
2640     void        *outary[1];
2642     int         result;
2644     /* Prepare parameters. */
2644     inary[0] = &imagecounter;
2646     outary[0] = &result;
2648     /* Invoke remote object method. */
2650     if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
2652                          RMIXIMGPROC_METHODS_UPDATEIMAGECOUNTER_INDEX, outary,
2652                          1, inary, 1))
2654     {
2654         int errno2 = errno;
2654         RMX_LOG((RMX_WARN(unable to invoke remote object method)))
2656         errno = errno2;
2656         return -1;
2658     }
2660     return 0;
2662 }
2664 /*
2664  * Update the successor entry of the predecessor plug-in and
2666  * returns the image counter (client-side method stub).
2668  *
2668  * \param  *remoteobj    ID of the remote object
2668  * \param  *successor    New successor entry
2670  * \param  *imagecounter The returned image counter from the predecessor
2670  * \return          0 on success or -1 on any error
2672  */
2674 #undef __FUNC__
2674 #define __FUNC__ "rmiximgprocclient_updatepredecessor"
2676 int rmiximgprocclient_updatepredecessor( rmix_remoteref_t *remoteobj,
2676                                         char            *successor,
2676                                         unsigned int    *imagecounter)
2678 {
2680     const void *inary[1];
2682     void        *outary[2];
2682     int         result;
2684     /* Prepare parameters. */
2686     inary[0] = successor;
2688     outary[0] = &result;
2688     outary[1] = &(*imagecounter);
2690     /* Invoke remote object method. */
2692     if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
2694                          RMIXIMGPROC_METHODS_UPDATEPREDECESSOR_INDEX, outary,
2694                          2, inary, 1))
2696     {
```

A. Appendix

```
2696     int errno2 = errno;
2698     RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
2699     errno = errno2;
2700     return -1;
2701 }
2702 return 0;
2703 }
2704
2705 /*
2706 * Update the predecessor entry of the successor plug-in (client-side method
2707 * stub).
2708 *
2709 * \param  *remoteobj  ID of the remote object
2710 * \param  *predecessor  New predecessor entry
2711 * \return  0 on success or -1 on any error
2712 */
2713 #undef __FUNC__
2714 #define __FUNC__ "rmiximgprocclient_updatesuccessor"
2715 int rmiximgprocclient_updatesuccessor( rmix_remoteref_t *remoteobj,
2716                                       char *predecessor)
2717 {
2718
2719     const void *inary[1];
2720
2721     void *outary[1];
2722     int result;
2723
2724     /* Prepare parameters. */
2725     inary[0] = predecessor;
2726
2727     outary[0] = &result;
2728
2729     /* Invoke remote object method. */
2730     if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
2731                          RMIXIMGPROC_METHODS_UPDATESUCCESSOR_INDEX, outary,
2732                          1, inary, 1))
2733     {
2734         int errno2 = errno;
2735         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
2736         errno = errno2;
2737         return -1;
2738     }
2739
2740     return 0;
2741 }
2742
2743 /*
2744 * Accepts calls for repairing the pipeline in case of an error. (server-side
2745 * stub)
2746 *
2747 * \param  object  The local object.
2748 * \param  outary  The output values array. The values array and its
2749 *                 containing values are allocated before the call with the
2750 *                 exception of variable arrays and strings. They are allocated
2751 *                 dynamically or explicitly set to NULL by this function. In
2752 *                 the case of variable arrays, the length value is allocated
2753 *                 before the call and is set to 0 if its variable array is
2754 *                 NULL.
2755 * \param  outcnt  The output values count.
2756 * \param  inary  The input values array. The values array and its containing
```

A. Appendix

```

    *          values are not modified or deallocated by this call.
2760 * \param incnt  The input values count.
    * \return      0 on success or -1 on any error.
2762 */
#undef __FUNC__
2764 #define __FUNC__ "rmixppm_repairpipe_call"
    RMX_METHOD_CALL(rmixppm_repairpipe_call)
2766 {
    int    result = 0;
2768
    char *node;          /* predecessor plug-in */
2770
    #ifdef DEBUG
2772     /* Check object parameter. */
    /*
2774     if (NULL == object)
    {
2776         errno = EINVAL;
            RMX_LOG((RMIX_WARN(object parameter is null)))
2778         errno = EINVAL;
            return -1;
2780     }
    */
2782     /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
2784     {
        errno = EINVAL;
            RMX_LOG((RMIX_WARN(outary parameter is null)))
            errno = EINVAL;
            return -1;
2788     }
2790
    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
2792     {
        errno = EINVAL;
            RMX_LOG((RMIX_WARN(outcnt parameter is zero)))
            errno = EINVAL;
            return -1;
2798     }
2800
    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
2802     {
        errno = EINVAL;
            RMX_LOG((RMIX_WARN(inary parameter is null)))
            errno = EINVAL;
            return -1;
2806     }
2808
    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
2810     {
        errno = EINVAL;
            RMX_LOG((RMIX_WARN(incnt paramet is zero)))
            errno = EINVAL;
            return -1;
2814     }
2816 #endif /* DEBUG */
2818
    /* Prepare input. */
2820     node = (char *)inary[0];

```


A. Appendix

```
2822     /* Call method function. */
2824     result = rmixppm_repairpipe( node);

2826     /* Prepare output. */
2828     /* Reset errno if needed. */
2830     if (0 != result)
2832     {
2834         errno = 0;
2836     }
2838     return 0;
2840 }

2842 /*
2844  * Tries to repair the pipeline in case of an error.
2846  *
2848  * \param  *node  Node which cannot be accessed
2850  * \return      0 on success or -1 on any error.
2852  */
2854 #undef __FUNC__
2856 #define __FUNC__ "rmixppm_repairpipe"
2858 int rmixppm_repairpipe (char *node)
2860 {
2862     int error;
2864     int ret;
2866     int i;
2868     int position;
2870     unsigned int    imagecounter;
2872     configlist_t    *tmppptrnodes;
2874     configlist_t    *plugin;
2876     rmix_remoteref_t *remoteobj;
2878     rmix_remoteref_t *remoteobj_pre; /* reference to the predecessor plug-in */
2880     rmix_remoteref_t *remoteobj_suc; /* reference to the successor plug-in */

2882     /* Lock ppm pipe mutex. */
2884     if (0 != (error = pthread_mutex_lock(&ppm_data.pipemutex)))
2886     {
2888         errno = error;
2890         PPM_PRINT((
2892             PPM_WARN(unable to lock ppm plug-in mutex))
2894             harness_syserr());
2896         return -1;
2898     }

2899     PPM_PRINT((
2901         PPM_INFO(*****)))
2903     PPM_PRINT((
2905         PPM_INFO(repair pipeline function called missing node = %s),node))

2907     /* search for the node in the list of used nodes
2909      * if the node is not in the list, it is assumed that another plug-in has
2911      * already called the restoration function and the failure is corrected */
2913     position = configlist_find_elementposition( node, ppm_data.usednodes);
2915     if (position != -1)
2917     {
2919         PPM_PRINT((
2921             PPM_INFO(missing node %s found in list at position %d), node,
2923             position))

2925     }

2927     /* check for available nodes */
```

A. Appendix

```
2886     if (configlist_listsize( ppm_data.freenodes) != 0)
2887     {
2888         PPM_PRINT((
2889             PPM_INFO(%d free nodes available),
2890             configlist_listsize( ppm_data.freenodes)))
2891
2892         tmpptrnodes = ppm_data.freenodes;
2893
2894         /* get the name of the failed plug-in unit */
2895         ret = configlist_return_element( &plugin, position,
2896             ppm_data.dist_plugins);
2897
2898         if (ret != 0)
2899         {
2900             PPM_PRINT((
2901                 PPM_WARN(plug-in could not find plug-in to reload)))
2902
2903             /* Unlock ppm pipe mutex. */
2904             if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
2905             {
2906                 errno = error;
2907                 PPM_PRINT((
2908                     PPM_WARN(unable to unlock ppm plug-in mutex)))
2909                 harness_syserr();
2910                 return -1;
2911             }
2912             return -1;
2913         }
2914
2915         PPM_PRINT((
2916             PPM_INFO(name of the plug-in to reload is %s),
2917             plugin->name))
2918
2919         /* try to reload the failed plug-in component on one of the free
2920         nodes */
2921         ret = ppm_load_harnesskernel_distributed( &ppm_data.freenodes,
2922             plugin, NULL);
2923
2924         if (ret != 0)
2925         {
2926             PPM_PRINT((
2927                 PPM_WARN(plug-in could not be reloaded)))
2928
2929             /* Unlock ppm pipe mutex. */
2930             if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
2931             {
2932                 errno = error;
2933                 PPM_PRINT((
2934                     PPM_WARN(unable to unlock ppm plug-in mutex)))
2935                 harness_syserr();
2936                 return -1;
2937             }
2938             return -1;
2939         }
2940
2941         /* free memory */
2942         configlist_delete_list(&plugin);
2943
2944         /* update the node with the reloaded plug-in in the list of used
2945         nodes and delete from the list of free nodes */
2946         if ( (configlist_reset_entry( ppm_data.freenodes->name, position,
2947             ppm_data.usednodes) != 0)
2948             || (configlist_delete_entry( 0, &ppm_data.freenodes) != 0) )
2949         {
2950             PPM_PRINT((
```

A. Appendix

```
2948         PPM_WARN(plug-in could not add the new node to the list
                of used nodes)))
2950
2951     /* Unlock ppm pipe mutex. */
2952     if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
2953     {
2954         errno = error;
2955         PPM_PRINT((
2956             PPM_WARN(unable to unlock ppm plug-in mutex)))
2957         harness_syserr();
2958         return -1;
2959     }
2960     return -1;
2961 }
2962
2963 /* reinitialise the new loaded plug-in and its neighbours */
2964
2965 /* move to the position of the reloaded node */
2966 tmpptrnodes = ppm_data.usednodes;
2967 for ( i=0; i<position; i++)
2968     tmpptrnodes = tmpptrnodes->next;
2969
2970 /* prepare the remote reference */
2971 ppm_createremoteref( &remoteobj, "1002", tmpptrnodes->name);
2972 /* check if the reloaded is the first unit of the pipeline
2973    the first plug-in unit has to open the images */
2974 if(position==0)
2975 {
2976     ret = rmiximgprocclient_initpipeline( remoteobj, position,
2977                                           ppm_data.sourcedir,
2978                                           "nada",
2979                                           tmpptrnodes->next->name,
2980                                           "nada", ppm_data.ppmnode);
2981
2982     if (ret != 0)
2983     {
2984         PPM_PRINT((
2985             PPM_WARN(could not initialize pipeline)))
2986         rmix_remoteref_destroy( &remoteobj);
2987         /* Unlock ppm pipe mutex. */
2988         if (0 != (error = pthread_mutex_unlock(
2989                 &ppm_data.pipemutex)))
2990         {
2991             errno = error;
2992             PPM_PRINT((
2993                 PPM_WARN(unable to unlock ppm plug-in mutex)))
2994             harness_syserr();
2995             return -1;
2996         }
2997         return -1;
2998     }
2999
3000     /* prepare the remote reference of the successor */
3001     ppm_createremoteref( &remoteobj_suc, "1002",
3002                         tmpptrnodes->next->name);
3003     /* update the predecessor entry of the successor plug-in */
3004     ret = rmiximgprocclient_updatesuccessor( remoteobj_suc,
3005                                             tmpptrnodes->name);
3006     if (ret != 0)
3007     {
3008         PPM_PRINT((
3009             PPM_WARN(could not initialize pipeline)))
3010         rmix_remoteref_destroy( &remoteobj_suc);
3011         /* Unlock ppm pipe mutex. */

```

A. Appendix

```
3012         if (0 != (error = pthread_mutex_unlock(
3013             &ppm_data.pipemutex)))
3014         {
3015             errno = error;
3016             PPM_PRINT((
3017                 PPM_WARN(unable to unlock ppm plug-in mutex))
3018                 harness_syserr());
3019             return -1;
3020         }
3021     }
3022     return -1;
3023 }
3024 rmix_remoteref_destroy( &remoteobj_suc);
3025
3026 /* to prevent a possible image loss resend all images by
3027    reinvoking the pipeline */
3028 ret = rmixingprocclient_invokepipeline_oneway(remoteobj);
3029 if (ret != 0)
3030 {
3031     PPM_PRINT((
3032         PPM_WARN(could not reinvoke pipeline)))
3033     rmix_remoteref_destroy( &remoteobj);
3034     /* Unlock ppm pipe mutex. */
3035     if (0 != (error = pthread_mutex_unlock(
3036         &ppm_data.pipemutex)))
3037     {
3038         errno = error;
3039         PPM_PRINT((
3040             PPM_WARN(unable to unlock ppm plug-in mutex))
3041             harness_syserr());
3042         return -1;
3043     }
3044 }
3045 /* the last plug-in unit has to store the images */
3046 else if(position == (configlist_listsize(ppm_data.dist_plugins)-1))
3047 {
3048     ret = rmixingprocclient_initpipeline( remoteobj, position,
3049         "nada", ppm_data.targetdir,
3050         "nada",
3051         tmpptrnodes->prev->name,
3052         ppm_data.ppmnode);
3053
3054     if (ret != 0)
3055     {
3056         PPM_PRINT((
3057             PPM_WARN(could not initialize pipeline)))
3058         rmix_remoteref_destroy( &remoteobj);
3059         /* Unlock ppm pipe mutex. */
3060         if (0 != (error = pthread_mutex_unlock(
3061             &ppm_data.pipemutex)))
3062         {
3063             errno = error;
3064             PPM_PRINT((
3065                 PPM_WARN(unable to unlock ppm plug-in mutex))
3066                 harness_syserr());
3067             return -1;
3068         }
3069     }
3070 }
3071
3072 /* create the remote reference of the predecessor plug-in */
3073 ppm_createremoteref( &remoteobj_pre, "1002",
3074     tmpptrnodes->prev->name);
```

A. Appendix

```
3074
3076     /* update the successor entry of the predecessor plug-in and
3078     gets its image counter */
3080     ret = rmiximgprocclient_updatepredecessor( remoteobj_pre ,
3082     tmpptrnodes->name,
3084     &imagecounter);
3086
3088     if (ret != 0)
3090     {
3092         PPM_PRINT((
3094             PPM_WARN(could not initialize pipeline)))
3096         rmix_remoteref_destroy( &remoteobj_pre);
3098         /* Unlock ppm pipe mutex. */
3100         if (0 != (error = pthread_mutex_unlock(
3102             &ppm_data.pipemutex)))
3104         {
3106             errno = error;
3108             PPM_PRINT((
3110                 PPM_WARN(unable to unlock ppm plug-in mutex)))
3112             harness_syserr();
3114             return -1;
3116         }
3118         return -1;
3120     }
3122
3124     /* update the image counter of the new loaded pipeline unit */
3126     ret = rmiximgprocclient_updateimagecounter( remoteobj ,
3128     imagecounter);
3130
3132     if (ret != 0)
3134     {
3136         PPM_PRINT((
3138             PPM_WARN(could not initialize pipeline)))
3140         rmix_remoteref_destroy( &remoteobj);
3142         /* Unlock ppm pipe mutex. */
3144         if (0 != (error = pthread_mutex_unlock(
3146             &ppm_data.pipemutex)))
3148         {
3150             errno = error;
3152             PPM_PRINT((
3154                 PPM_WARN(unable to unlock ppm plug-in mutex)))
3156             harness_syserr();
3158             return -1;
3160         }
3162         return -1;
3164     }
3166
3168     /* trigger the successor plug-in to resend its internal backup
3170     list to prevent an image loss */
3172     ret = rmiximgprocclient_sendworklist_oneway (remoteobj_pre);
3174     if (ret != 0)
3176     {
3178         PPM_PRINT((
3180             PPM_WARN(could not initialize pipeline)))
3182         rmix_remoteref_destroy( &remoteobj_pre);
3184         /* Unlock ppm pipe mutex. */
3186         if (0 != (error = pthread_mutex_unlock(
3188             &ppm_data.pipemutex)))
3190         {
3192             errno = error;
3194             PPM_PRINT((
3196                 PPM_WARN(unable to unlock ppm plug-in mutex)))
3198             harness_syserr();
3200             return -1;
3202         }
3204     }
```

A. Appendix

```

    return -1;
3138 }
    rmix_remoteref_destroy( &remoteobj_pre);
3140
}
3142 /* every plug-in unit in between has to forward the images and
    acknowledgments */
3144 else
    {
3146     ret = rmiximgprocclient_initpipeline( remoteobj, position,
                                           "nada", "nada",
3148     tmpptrnodes->next->name,
                                           tmpptrnodes->prev->name,
3150     ppm_data.ppmnode);

    if (ret != 0)
3152     {
        PPM_PRINT((
3154     PPM_WARN(could not initialize pipeline)))
        rmix_remoteref_destroy( &remoteobj);
3156     /* Unlock ppm pipe mutex. */
        if (0 != (error = pthread_mutex_unlock(
3158     &ppm_data.pipemutex)))
            {
3160     errno = error;
            PPM_PRINT((
3162     PPM_WARN(unable to unlock ppm plug-in mutex)))
            harness_syserr();
3164     return -1;
            }
3166     return -1;
        }
3168

    /* create remote reference of the successor */
3170 ppm_createremoteref( &remoteobj_suc, "1002",
                       tmpptrnodes->next->name);
3172

    /* update the predecessor entry of the successor plug-in */
3174 ret = rmiximgprocclient_updatesuccessor( remoteobj_suc,
                                           tmpptrnodes->name);
3176
    if (ret != 0)
    {
3178     PPM_PRINT((
        PPM_WARN(could not initialize pipeline)))
3180     rmix_remoteref_destroy( &remoteobj_suc);
        /* Unlock ppm pipe mutex. */
3182     if (0 != (error = pthread_mutex_unlock(
        &ppm_data.pipemutex)))
            {
3184     errno = error;
            PPM_PRINT((
3186     PPM_WARN(unable to unlock ppm plug-in mutex)))
            harness_syserr();
3188     return -1;
            }
3190     return -1;
        }
3192
    }
    rmix_remoteref_destroy( &remoteobj_suc);
3194

    /* create the remote reference of the predecessor plug-in */
3196 ppm_createremoteref( &remoteobj_pre, "1002",
                       tmpptrnodes->prev->name);
3198

    /* update the successor entry of the predecessor plug-in and
```

A. Appendix

```
3200     gets its image counter */
3201     ret = rmiximgprocclient_updatepredecessor( remoteobj_pre ,
3202                                               tmpptrnodes->name,
3203                                               &imagecounter);
3204
3205     if (ret != 0)
3206     {
3207         PPM_PRINT((
3208             PPM_WARN(could not initialize pipeline)))
3209         rmix_remoteref_destroy( &remoteobj_pre);
3210         /* Unlock ppm pipe mutex. */
3211         if (0 != (error = pthread_mutex_unlock(
3212             &ppm_data.pipemutex)))
3213         {
3214             errno = error;
3215             PPM_PRINT((
3216                 PPM_WARN(unable to unlock ppm plug-in mutex)))
3217             harness_syserr();
3218             return -1;
3219         }
3220         return -1;
3221     }
3222
3223     /* set the image counter of the new loaded pipeline unit */
3224     ret = rmiximgprocclient_updateimagecounter( remoteobj ,
3225                                               imagecounter);
3226
3227     if (ret != 0)
3228     {
3229         PPM_PRINT((
3230             PPM_WARN(could not initialize pipeline)))
3231         rmix_remoteref_destroy( &remoteobj);
3232         /* Unlock ppm pipe mutex. */
3233         if (0 != (error = pthread_mutex_unlock(
3234             &ppm_data.pipemutex)))
3235         {
3236             errno = error;
3237             PPM_PRINT((
3238                 PPM_WARN(unable to unlock ppm plug-in mutex)))
3239             harness_syserr();
3240             return -1;
3241         }
3242         return -1;
3243     }
3244
3245     /* trigger the predecessor to resend its internal backup list
3246        to prevent a possible image loss */
3247     ret = rmiximgprocclient_sendworklist_oneway (remoteobj_pre);
3248     if (ret != 0)
3249     {
3250         PPM_PRINT((
3251             PPM_WARN(could not initialize pipeline)))
3252         rmix_remoteref_destroy( &remoteobj_pre);
3253         /* Unlock ppm pipe mutex. */
3254         if (0 != (error = pthread_mutex_unlock(
3255             &ppm_data.pipemutex)))
3256         {
3257             errno = error;
3258             PPM_PRINT((
3259                 PPM_WARN(unable to unlock ppm plug-in mutex)))
3260             harness_syserr();
3261             return -1;
3262         }
3263         return -1;
3264     }
3265 }
```

A. Appendix

```

    rmix_remoteref_destroy( &remoteobj_pre);
3264     }
3265     rmix_remoteref_destroy( &remoteobj);
3266
3267     }
3268     else
3269     {
3270         PPM_PRINT((
3271             PPM_INFO(no free node(s) available)))
3272     }
3273 }
3274 else
3275 {
3276     PPM_PRINT((
3277         PPM_INFO(missing node %s not found in list), node))
3278 }
3279
3280 PPM_PRINT((
3281     PPM_INFO(*****)))
3282
3283 /* Unlock ppm pipe mutex. */
3284 if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
3285 {
3286     errno = error;
3287     PPM_PRINT((
3288         PPM_WARN(unable to unlock ppm plug-in mutex)))
3289     harness_syserr();
3290     return -1;
3291 }
3292
3293 return 0;
3294 }
3295
3296
3297 /*
3298  * Sends again the worklist to the reloaded plug-in (client-side method stub).
3299  *
3300  * \param  *remoteobj  ID of the remote object
3301  * \return  0 on success or -1 on any error
3302  */
3303 #undef __FUNC__
3304 #define __FUNC__ "rmiximprocclient_sendworklist_oneway"
3305 int rmiximprocclient_sendworklist_oneway (rmix_remoteref_t *remoteobj)
3306 {
3307     /* Invoke remote object method. */
3308     if (0 != rmix_oneway(remoteobj, &rmiximprocclient_interface,
3309         RMIXIMPROC_METHODS_SENDWORKLIST_INDEX, NULL, 0))
3310     {
3311         int errno2 = errno;
3312         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
3313         errno = errno2;
3314         return -1;
3315     }
3316
3317     return 0;
3318 }
3319
3320
3321 /*****
3322  *
3323  * END OF FILE
3324  *
3325  *****/

```


A.3.2.3. Header File for the PPM Utility Library

```

1  /*****
   *
3  * Header file for the plug-in loader utils module.
   * Copyright (c) Ronald Baumann
5  *
   * For more information see the following files in the source distribution top-
7  * level directory or package data directory (usually /usr/local/share/package):
   *
9  * - README    for general package information.
   * - INSTALL  for package install information.
11 * - COPYING   for package license information and copying conditions.
   * - AUTHORS   for package authors information.
13 * - ChangeLog for package changes information.
   *
15 *****/

17 /** \file readconf.h
   * \brief Header file for plug-in loader utils library.
19 *
   * The plug-in loader utils library contains functions which are used by the
21 * parallel plug-in manager.
   * It supports the reading of a configuration file and the storing of the read
23 * data in lists.
   */
25

27 /* Avoid to include the content of this header file twice. */
   #ifndef READCONF_READCONF_H
29 #define READCONF_READCONF_H

31

   /*****
33 *
   * Macros
35 *
   *****/

37 /** \def READCONF_QUOTES(string)
39 * \brief Quoting a string.
   */
41 #define READCONF_QUOTES(string) #string

43

45 /** \def READCONF_STRING(string)
   * \brief A string.
   */
47 #define READCONF_STRING(string) READCONF_QUOTES(string)

49

51 /** \def READCONF_FUSE(arg1, arg2)
   * \brief Fuse two strings.
   */
53 #define READCONF_FUSE(arg1, arg2) arg1##arg2

55

57 /** \def READCONF_JOIN(arg1, arg2)
   * \brief Joining two text constants.
   */
59 #define READCONF_JOIN(arg1, arg2) READCONF_FUSE(arg1, arg2)

61

```

A. Appendix

```
/** \def READCONF_WARN(string)
63 * \brief Debug printout.
*/
65 #define READCONF_WARN(string) \
    fprintf(stderr, \
67         READCONF_STRING(warn: libplutils:%d:%s:%u:%s: string\n), \
            getpid(), __FILE__, __LINE__, __FUNC__
69

71 /** \def READCONF_INFO(string)
    * \brief Debug printout.
73 */
#define READCONF_INFO(string) \
75     fprintf(stderr, \
        READCONF_STRING(info: libplutils:%d:%s:%u:%s: string\n), \
77         getpid(), __FILE__, __LINE__, __FUNC__

79
/** \def READCONF_PRINT(string)
81 * \brief Wrapper for debug printout.
*/
83 #ifdef DEBUG
#define READCONF_PRINT(string) string);
85 #else /* DEBUG */
#define READCONF_PRINT(string)
87 #endif /* DEBUG */

89
/** \def FREE(x)
91 * \brief Macro frees allocated memory and add the NULL pointer.
    *
93 * A NULL pointer will be added after freeing allocated memory.
*/
95 #define FREE(x) {if (x != NULL) {free(x); x = NULL;}}

97
/** \def CONFFILE
99 * \brief Name of the configuration file.
*/
101 #define CONFFILE "/home/ronald/paralleplugins/data/config.conf"

103
/** \def NODES
105 * \brief Node section of a configuration file is converted into a linked list.
*/
107 #define NODES 0

109
/** \def REPLICATED
111 * \brief Section with names of replicated plug-ins in a configuration file is
    * converted into a linked list.
113 */
#define REPLICATED 1
115

117 /** \def DISTRIBUTED
    * \brief Section with names of distributed plug-ins in a configuration file
119 * is converted into a linked list.
*/
121 #define DISTRIBUTED 2

123
/** \def INPUT
```

A. Appendix

```
125 * \brief Input section in a configuration file is converted into a linked
    * list.
127 */
#define INPUT 3
129
131 /** \def NODESECTION
    * \brief Name of the node section in the configuration file.
133 */
#define NODESECTION "nodelist"
135
137 /** \def NODEENTRY
    * \brief Name of the entry in the configuration file node section.
139 */
#define NODEENTRY "addresses"
141
143 /** \def RSECTION
    * \brief Name of the replicated plug-in section in the configuration file.
145 */
#define RSECTION "replicated"
147
149 /** \def RENTRY
    * \brief Name of the entry in the configuration file replicated section.
151 */
#define RENTRY "plugins"
153
155 /** \def DSECTION
    * \brief Name of the distributed plug-in section in the configuration file.
157 */
#define DSECTION "distributed"
159
161 /** \def DENTRY
    * \brief Name of the entry in the configuration file distributed section.
163 */
#define DENTRY "plugins"
165
167 /** \def INPUTSECTION
    * \brief Name of the input section in the configuration file.
169 */
#define INPUTSECTION "input"
171
173 /** \def INPUTENTRY
    * \brief Name of the entry in the configuration file input section.
175 */
#define INPUTENTRY "files"
177
179 /** \def PPMODE
    * \brief Selection parallel plug-in mode (1) or distributed parallel plug-in
181 * mode (0).
    */
183 #define PPMODE "mode"
185
187 /** \def PPMODE
    * \brief Node of the Parallel Plug-in Manager.
```

A. Appendix

```
189 */
#define PPMNODE      "ppmnode"

191
192 /*****
193 *
194 * Includes
195 *
196 *****/
197 #include <stdio.h>
198 #include <stdlib.h>
199 #include <string.h>
200 #include <sys/types.h>
201 #include <unistd.h>
202 #include <confuse.h>
203
204 /*****
205 *
206 * Data Types
207 *
208 *****/
209
210 /** \struct configlist_t
211 * \brief Element of the linked list for storing configuration information.
212 *
213 * Each element contains a name for storing ip-addresses of hosts or the name
214 * of a plug-in and a pointer to the next and previous element in the list.
215 */
216 #typedef struct configlist
217 {
218     char *name; /* ip-address or name of plug-in */
219     struct configlist *next; /* pointer to the next list element */
220     struct configlist *prev; /* pointer to the prev list element */
221 } configlist_t;
222
223
224 /*****
225 *
226 * Function Prototypes
227 *
228 *****/
229
230 /** \fn int configuration_read_file( configlist_t **nodelist, \
231                                     configlist_t **replicatedlist, \
232                                     configlist_t **distributedlist, \
233                                     configlist_t **inputlist \
234                                     int *mode, \
235                                     char **ppmnode);
236
237 * \brief Reads the config file, information is stored in linked lists.
238 *
239 * \param **nodelist List pointer for the node name(s)
240 * \param **replicatedlist List pointer for replicated plug-in name(s)
241 * \param **distributedlist List pointer for distributed plug-in name(s)
242 * \param **inputlist List pointer for input file(s)
243 * \param *mode Replicated or distributed parallel plug-in mode
244 * \param **ppmnode Node of the PPM
245 * \return 0 on success or -1 on any error
246 */
247 int configuration_read_file( configlist_t **nodelist,
248                             configlist_t **replicatedlist,
249                             configlist_t **distributedlist,
250                             configlist_t **inputlist,
```

A. Appendix

```
251             int          *mode,
                char        **ppmnode);
253
255 /** \fn      int configuration_validate_nodelist( cfg_t      *cfg, \
                cfg_opt_t *opt);
257 * \brief    Validates the node section in the configuration file.
                *
259 * \param    *cfg Pointer to the configuration file structure
                * \param    *opt Name of the section to validate
261 * \return   0 on success or -1 on any error
                */
263 int configuration_validate_nodelist( cfg_t      *cfg,
                cfg_opt_t *opt);
265
267 /** \fn      int configlist_insert_element( char          *element, \
                configlist_t **list_pointer);
269 * \brief    Insert an element in a list.
                *
271 * \param    *element      New list element name
                * \param    **list_pointer Pointer to the linked list
273 * \return   0 on success or -1 on any error
                */
275 int configlist_insert_element( char          *element,
                configlist_t **list_pointer);
277
279 /** \fn      void configuration_conver2list( cfg_t          *cfg, \
                configlist_t **list_pointer, \
281             int          type);
                * \brief    Converts a section of the configuration file structure into a list.
283 *
                * \param    *cfg      Pointer to the structure of the configuration file
285 * \param    **list_pointer Pointer to the new linked list
                * \param    type     Defines the section of the configuration file
287 * \return   0 on success or -1 on any error
                */
289 int configuration_conver2list( cfg_t          *cfg,
                configlist_t **list_pointer,
291             int          type);
293
295 /** \fn      void configlist_delete_list( configlist_t **list_pointer);
                * \brief    Deletes the linked list defined by the pointer.
                *
297 * \param    **list_pointer Pointer to the linked list
                */
299 void configlist_delete_list( configlist_t **list_pointer);
301
303 /** \fn      void configlist_print_list( configlist_t *list_pointer);
                * \brief    Prints the list elements.
                *
305 * \param    *list_pointer Pointer to the linked list
                */
307 void configlist_print_list( configlist_t *list_pointer);
309
311 /** \fn      int configlist_listsize( configlist_t *list_pointer);
                * \brief    Return the number of list elements.
                *
313 * \param    *list_pointer Pointer to the linked list
```

A. Appendix

```

    * \return          Number of list elements
315 */
int configlist_listsize( configlist_t *list_pointer);
317

319 /** \fn    int configlist_check_entries( configlist_t *list_pointer);
    * \brief  Checks if all names entries != NULL.
321 *
    * \param  *list_pointer Pointer to the linked list
323 * \return          0 if all name entries != NULL otherwise -1
    */
325 int configlist_check_entries( configlist_t *list_pointer);

327
/** \fn    int configlist_find_elementposition( char          *entry, \
329                                               configlist_t *list_pointer);
    * \brief  Finds the position of an entry.
331 *
    * \param  *entry      Entry to find
333 * \param  *list_pointer Pointer to the linked list
    * \return          The position otherwise -1
335 */
int configlist_find_elementposition( char          *entry,
337                                   configlist_t *list_pointer);

339
/** \fn    int configlist_return_element( configlist_t **element, \
341                                       int          position, \
                                       configlist_t *list_pointer);
343 * \brief  Returns an element found at the specific position
    *
345 * \param  **element    Found list element
    * \param  position    Position of the element in the list
347 * \param  *list_pointer Pointer to the linked list
    * \return          0 on success otherwise -1
349 */
int configlist_return_element( configlist_t **element,
351                             int          position,
                             configlist_t *list_pointer);
353

355 /** \fn    int configlist_reset_entry( char          *entry, \
357                                       int          position, \
                                       configlist_t *list_pointer);
    * \brief  Sets the entry at the defined position with the new value.
359 *
    * \param  *char      New value of the entry
361 * \param  position    Position of the entry
    * \param  *list_pointer Pointer to the linked list
363 * \return          0 on success otherwise -1
    */
365 int configlist_reset_entry( char          *entry,
                             int          position,
                             configlist_t *list_pointer);
367

369
/** \fn    int configlist_delete_entry( int          position, \
371                                       configlist_t **list_pointer);
    * \brief  Deletes the element at the position.
373 *
    * \param  *int      Position of the element.
375 * \param  **list_pointer Pointer to the linked list
    * \return          The position otherwise -1

```

A. Appendix

```
377 */
379 int configlist_delete_entry( int      position ,
                             configlist_t **list_pointer);
381
382 #endif
383 /*****
384 *
385 * END OF FILE
386 *
387 *****/
```

A.3.2.4. Source File for the PPM Utility Library

```
1 /*****
2 *
3 * Source file for the Integral-Loader module.
4 * Copyright (c) Ronald Baumann.
5 *
6 * For more information see the following files in the source distribution top-
7 * level directory or package data directory (usually /usr/local/share/package):
8 *
9 * - README    for general package information.
10 * - INSTALL   for package install information.
11 * - COPYING   for package license information and copying conditions.
12 * - AUTHORS   for package authors information.
13 * - ChangeLog for package changes information.
14 *
15 *****/
16
17 /** \file readconf.c
18 * \brief Source file for plug-in loader utils library.
19 *
20 * The plug-in loader utils library contains functions which are used by
21 * parallel plug-in manager.
22 * It supports the reading of a configuration file and the storing of the read
23 * data in lists.
24 */
25
26 /*****
27 *
28 * Includes
29 *
30 *****/
31 #include "readconf.h"
32
33
34 /*****
35 *
36 * Functions
37 *
38 *****/
39
40 /*
41 * Reads the config file , information is stored in linked lists.
42 *
43 * \param **nodelist      List pointer for the node name(s)
44 * \param **replicatedlist List pointer for replicated plug-in name(s)
45 * \param **distributedlist List pointer for distributed plug-in name(s)
46 * \param **inputlist     List pointer for input file(s)
47 * \param *mode            Replicated or distributed parallel plug-in mode
48 */
```

A. Appendix

```
49 * \param **ppmnode      Node of the PPM
   * \return              0 on success or -1 on any error
51 */
#define __FUNC__
53 #define __FUNC__ "configuration_read_file"
int configuration_read_file( configlist_t **nodelist,
55                          configlist_t **replicatedlist,
                          configlist_t **distributedlist,
57                          configlist_t **inputlist,
                          int *mode,
59                          char **ppmnode)
{
61     /* defines the node section in the conf file */
    cfg_opt_t nodelist_opts[] =
63     {
        CFG_STR_LIST( NODEENTRY, , CFGF_NONE),
65         CFG_END()
    };
67     /* defines the replicated plug-in section in the conf file */
    cfg_opt_t replicated_opts[] =
69     {
        CFG_STR_LIST( RENTRY, , CFGF_NONE),
71         CFG_END()
    };
73     /* defines the distributed plug-in section in the conf file */
    cfg_opt_t distributed_opts[] =
75     {
        CFG_STR_LIST( DENTRY, , CFGF_NONE),
77         CFG_END()
    };
81     /* defines the input section in the conf file */
    cfg_opt_t input_opts[] =
83     {
        CFG_STR_LIST( INPUTENTRY, , CFGF_NONE),
85         CFG_END()
    };
87     /* build the conf file structure */
    cfg_opt_t opts[] =
91     {
        CFG_SEC( NODESECTION, nodelist_opts, CFGF_NONE),
93         CFG_SEC( RSECTION, replicated_opts, CFGF_NONE),
        CFG_SEC( DSECTION, distributed_opts, CFGF_NONE),
95         CFG_SEC( INPUTSECTION, input_opts, CFGF_NONE),
        CFG_INT( PPMODE, 0, CFGF_NONE),
97         CFG_STR( PPMODE, NULL, CFGF_NONE),
        CFG_END()
99     };
101     cfg_t *cfg;
103     READCONF_PRINT((
        READCONF_INFO(start reading configuration file)))
105     /* initialize the conf file structure */
107     cfg = cfg_init(opts, CFGF_NONE);
109     /* initialize the validation function */
    cfg_set_validate_func(cfg, NODESECTION, configuration_validate_nodelist);
111
```


A. Appendix

```
113  /* read the conf file */
114  if (cfg_parse(cfg, CONFFILE) == CFG_PARSE_ERROR)
115      return -1;
116
117  /* convert the conf file section to lists */
118  if (configuration_conver2list( cfg, nodelist, NODES) == -1)
119      return -1;
120  if (configuration_conver2list( cfg, replicatedlist, REPLICATED) == -1)
121      return -1;
122  if (configuration_conver2list( cfg, distributedlist, DISTRIBUTED) == -1)
123      return -1;
124  if (configuration_conver2list( cfg, inputlist, INPUT) == -1)
125      return -1;
126
127  /* read the node, on which the PPM runs */
128  *mode = cfg_getint(cfg, PPMODE);
129
130  *ppmnode = (char*)malloc( sizeof(char) * (strlen(cfg_getstr(cfg, PPMODE))
131      + 1));
132  if ((*ppmnode) == NULL)
133  {
134      READCONF_PRINT((
135          READCONF_WARN(could not allocate memory)))
136      return -1;
137  }
138  strcpy( (*ppmnode), cfg_getstr(cfg, PPMODE));
139
140  /* delete the conf file structure */
141  cfg_free(cfg);
142  return 0;
143 }
144
145 /*
146  * Validates the node section in the configuration file.
147  *
148  * \param *cfg Pointer to the configuration file structure
149  * \param *opt Name of the section to validate
150  * \return 0 on success or -1 on any error
151 */
152 #undef __FUNC__
153 #define __FUNC__ "configuration_validate_nodelist"
154 int configuration_validate_nodelist( cfg_t *cfg,
155     cfg_opt_t *opt)
156 {
157     /* get the last parsed section */
158     cfg_t *sec = cfg_opt_getnsec(opt, cfg_opt_size(opt) - 1);
159
160     /* validate that a "string" option is set in the section */
161     if (cfg_size(sec, NODEENTRY) > 0 && cfg_getstr(sec, NODEENTRY) != NULL)
162         return CFG_SUCCESS;
163
164     /* otherwise the option is not set at all, log a message and return error */
165     cfg_error(cfg, "missing addresses option in nodelist section");
166     READCONF_PRINT((
167         READCONF_WARN(missing addresses option in nodelist section of
168             configuration file)))
169     return CFG_PARSE_ERROR;
170 }
171
172 /*
173  * Converts a section of the configuration file structure into a list.
```

A. Appendix

```
175 *
176 * \param *cfg Pointer to the structure of the configuration file
177 * \param **list_pointer Pointer to the new linked list
178 * \param type Defines the section of the configuration file
179 * \return 0 on success or -1 on any error
180 */
181 #undef __FUNC__
182 #define __FUNC__ "configuration_conver2list"
183 int configuration_conver2list( cfg_t *cfg,
184                               configlist_t **list_pointer,
185                               int type)
186 {
187     cfg_t *section;
188     int i;
189     int ret;
190
191     /* which section is converted? */
192     switch(type)
193     {
194     case(NODES):
195         {
196             section = cfg_getsec(cfg, NODESECTION);
197             /* control if the section contains information */
198             if (cfg_getstr(section, NODEENTRY) == NULL)
199             {
200                 (*list_pointer) = NULL;
201                 return -1;
202             }
203             /* create list */
204             for( i = 0; i < cfg_size(section, NODEENTRY); i++)
205             {
206                 ret = configlist_insert_element(
207                     cfg_getnstr(section, NODEENTRY, i), list_pointer);
208                 if (ret == -1)
209                     return -1;
210             }
211             break;
212         }
213     case(REPLICATED):
214         {
215             section = cfg_getsec(cfg, RSECTION);
216             /* control if the section contains information */
217             if (cfg_getstr(section, RENTRY) == NULL)
218             {
219                 (*list_pointer) = NULL;
220                 return -1;
221             }
222             /* create list */
223             for( i = 0; i < cfg_size(section, RENTRY); i++)
224             {
225                 ret = configlist_insert_element(
226                     cfg_getnstr(section, RENTRY, i), list_pointer);
227                 if (ret == -1)
228                     return -1;
229             }
230             break;
231         }
232     case(DISTRIBUTED):
233         {
234             section = cfg_getsec(cfg, DSECTION);
235             /* control if the section contains information */
236             if (cfg_getstr(section, DENTRY) == NULL)
237             {
```

A. Appendix

```

239     (*list_pointer) = NULL;
        return -1;
    }
241     /* create list */
    for( i = 0; i < cfg_size(section , DENTRY); i++)
243     {
        ret = configlist_insert_element(
245             cfg_getnstr(section , DENTRY, i), list_pointer);
        if (ret == -1)
247             return -1;
    }
249     break;
}
251 case(INPUT):
{
253     section = cfg_getsec(cfg , INPUTSECTION);
    /* control if the section contains information */
255     if (cfg_getstr(section , INPUTENTRY) == NULL)
    {
257         (*list_pointer) = NULL;
            return -1;
    }
259     /* create list */
    for( i = 0; i < cfg_size(section , INPUTENTRY); i++)
261     {
        ret = configlist_insert_element(
263             cfg_getnstr(section , INPUTENTRY, i), list_pointer);
265
        if (ret == -1)
267             return -1;
    }
269     break;
}
271 }
return 0;
273 }

275
277 /* Inserts an element in a list.
    *
279 * \param *element      New list element name
    * \param **list_pointer Pointer to the linked list
281 * \return              0 on success or -1 on any error
    */
283 #undef __FUNC__
#define __FUNC__ "configlist_insert_element"
285 int configlist_insert_element( char *element,
                                configlist_t **list_pointer)
287 {
    configlist_t *new_element;
289     configlist_t *tmp_element;

291     /* allocate memory for the new element */
    new_element = (configlist_t*)malloc(sizeof(configlist_t));
293     if ( new_element == NULL )
    {
295         READCONF_PRINT((
            READCONF_WARN(allocating of memory for new list element not
297             possible)))
        return -1;
299     }

```

A. Appendix

```
301  /* allocate memory for the name string of the new element */
new_element->name = (char*)malloc(sizeof(char)*(strlen(element)+1));
303  if ( new_element->name == NULL )
{
305      READCONF_PRINT((
          READCONF_WARN(allocating of memory for the name string of the new
307          list element not possible))
          return -1;
309  }

311  /* store the information */
strcpy( new_element->name, element);
313  new_element->next = NULL;

315  /* add the new element at the end of the list */
317  if ( *list_pointer == NULL )
{
319      *list_pointer = new_element;
new_element->next = NULL;
321      new_element->prev = NULL;
}
323  else
{
325      tmp_element = *list_pointer;

327      while ( tmp_element->next != NULL )
{
329          tmp_element = tmp_element->next;
}

331      tmp_element->next = new_element;
333      new_element->prev = tmp_element;
new_element->next = NULL;
335  }

337  return 0;
}
339

341  /*
* Prints the list elements.
343  *
* \param *list_pointer Pointer to the linked list
345  */
#ifdef __FUNC__
347  #define __FUNC__ "configlist_print_list"
void configlist_print_list( configlist_t *list_pointer)
349  {
    configlist_t *current_element;

351
    current_element = list_pointer;
353    while ( current_element != NULL )
    {
355        READCONF_PRINT((
            READCONF_INFO(name = %s\n), current_element->name))
357
            current_element = current_element->next;
359    }
}
361

363  /*
```

A. Appendix

```

    * Deletes the linked list defined by the pointer.
365 *
    * \param **list_pointer Pointer to the linked list
367 */
#undef __FUNC__
369 #define __FUNC__ "configlist_delete_list"
void configlist_delete_list( configlist_t **list_pointer)
371 {
    configlist_t *tmp_pointer;
373
    while( *list_pointer != NULL )
375 {
        tmp_pointer = *list_pointer;
377         *list_pointer = (*list_pointer)->next;

379         /* delete name of the element */
        FREE(tmp_pointer->name);
381
        /* delete element */
383         FREE(tmp_pointer);
    }
385     *list_pointer = NULL;
387 }

389 /*
    * Return the number of list elements.
391 *
    * \param *list_pointer Pointer to the linked list
393 * \return      Number of list elements
    */
395 #undef __FUNC__
#define __FUNC__ "configlist_listsize"
397 int configlist_listsize( configlist_t *list_pointer)
    {
399     configlist_t *current_element;
        int counter = 0;
401
        current_element = list_pointer;
403     while ( current_element != NULL )
        {
405         current_element = current_element->next;
            counter ++;
407     }
        return counter;
409 }

411
413 /*
    * Checks if all names entries != NULL.
    *
415 * \param *list_pointer Pointer to the linked list
    * \return      0 if all name entries != NULL otherwise -1
417 */
#undef __FUNC__
419 #define __FUNC__ "configlist_check_entries"
int configlist_check_entries( configlist_t *list_pointer)
421 {
    configlist_t *current_element;
423
    current_element = list_pointer;
425     while ( current_element != NULL )
        {
```

A. Appendix

```
427     if ( current_element->name == NULL )
428         return -1;
429     current_element = current_element->next;
430 }
431 return 0;
432 }
433
434 /*
435  * Finds the position of an entry.
436  *
437  * \param *char      Entry to find
438  * \param *list_pointer Pointer to the linked list
439  * \return          The position or otherwise -1
440  */
441 #undef __FUNC__
442 #define __FUNC__ "configlist_find_elementposition"
443 int configlist_find_elementposition( char *entry,
444                                     configlist_t *list_pointer)
445 {
446     configlist_t *current_element;
447     int position = 0;
448
449     current_element = list_pointer;
450     while ( current_element != NULL )
451     {
452         if ( strcmp( current_element->name, entry ) == 0 )
453             return position;
454         current_element = current_element->next;
455         position++;
456     }
457
458     return -1;
459 }
460
461 /*
462  * Returns an element found at the specific position
463  *
464  * \param **element   Found list element
465  * \param position    Position of the element in the list
466  * \param *list_pointer Pointer to the linked list
467  * \return            0 on success otherwise -1
468  */
469 #undef __FUNC__
470 #define __FUNC__ "configlist_return_element"
471 int configlist_return_element( configlist_t **element,
472                               int position,
473                               configlist_t *list_pointer)
474 {
475     configlist_t *current_element;
476     int i;
477
478     current_element = list_pointer;
479
480     /* check if the position is within the list size */
481     if ( (configlist_listsize(current_element) < position) || (position < 0) )
482     {
483         READCONF_PRINT((
484             READCONF_WARN(position exceeds list size)))
485         return -1;
486     }
487 }
488
489
```

A. Appendix

```
/* move forward to position */
491 for ( i=0; i<position; i++)
    {
493     current_element = current_element->next;
    }
495
(*element) = (configlist_t*)malloc( sizeof(configlist_t));
497 if ( (*element) == NULL )
    {
499     READCONF_PRINT((
501         READCONF_WARN(allocating of memory for found list element not
                    possible)))
        return -1;
503     }
505
/* allocate memory for the name string of the element */
(*element)->name =(char*)malloc( sizeof(char) *
507     (strlen(current_element->name) + 1));
if ( (*element)->name == NULL )
509     {
        READCONF_PRINT((
511         READCONF_WARN(allocating of memory for the name string of the
                    list element not possible)))
        return -1;
513     }
515
/* store the information */
517 strcpy( (*element)->name, current_element->name);
(*element)->next = NULL;
519
return 0;
521 }

523
/*
525 * Sets the entry at the defined position with the new value.
    *
527 * \param *char      New value of the entry
    * \param position  Position of the entry
529 * \param *list_pointer Pointer to the linked list
    * \return          0 on success otherwise -1
531 */
#define __FUNC__
533 #define __FUNC__ "configlist_reset_entry"
int configlist_reset_entry( char *entry,
535     int position,
                    configlist_t *list_pointer)
537 {
    configlist_t *tmp_pointer;
539     int i;

541     tmp_pointer = list_pointer;

543     /* check if position is within the list size */
    if ( (configlist_listsize(tmp_pointer) < position) || (position < 0) )
545     {
        READCONF_PRINT((
547         READCONF_WARN(position exceeds list size)))
        return -1;
549     }

551     tmp_pointer = list_pointer;
```

A. Appendix

```
553     for (i=0; i<position; i++)
554         tmp_pointer = tmp_pointer->next;
555
556     FREE(tmp_pointer->name);
557
558     /* allocate memory for the name string of the new element */
559     tmp_pointer->name = (char*)malloc(sizeof(char)*(strlen(entry)+1));
560     if ( tmp_pointer->name == NULL )
561     {
562         READCONF_PRINT((
563             READCONF_WARN(allocating of memory for the name string not
564                 possible)))
565         return -1;
566     }
567
568     /* store the information */
569     strcpy( tmp_pointer->name, entry);
570
571     return 0;
572 }
573
574 /*
575  * Deletes the element at the position.
576  *
577  * \param  *int          Position of the element.
578  * \param  **list_pointer Pointer to the linked list
579  * \return          0 on success otherwise -1
580  */
581 #undef __FUNC__
582 #define __FUNC__ "configlist_delete_entry"
583 int configlist_delete_entry( int          position ,
584                             configlist_t **list_pointer)
585 {
586     configlist_t *tmp_pointer;
587     configlist_t *del_element;
588     int i=0;
589
590     tmp_pointer = *list_pointer;
591
592     /* check if the position is within the list size */
593     if ( (configlist_listsize(tmp_pointer) <= position) || (position < 0) )
594     {
595         READCONF_PRINT((
596             READCONF_WARN(position exceeds list size)))
597         return -1;
598     }
599
600     /* if it is the first element in the list change the pointer to the list*/
601     if ( position == 0 )
602     {
603         tmp_pointer = *list_pointer;
604         *list_pointer = (*list_pointer)->next;
605         if ((*list_pointer) != NULL)
606             (*list_pointer)->prev = NULL;
607
608         /* delete name of the element */
609         FREE(tmp_pointer->name);
610
611         /* delete element */
612         FREE(tmp_pointer);
613     }
614     else
```


A. Appendix

```
617 {
    tmp_pointer = *list_pointer;
619     for (i=0; i<position-1; i++)
        tmp_pointer = tmp_pointer->next;
621     del_element = tmp_pointer->next;
623     /* if it was the last element in the list */
625     if (del_element->next == NULL)
        {
627         tmp_pointer->next = NULL;
        }
629     else
        {
631         tmp_pointer->next = del_element->next;
        del_element->next->prev = tmp_pointer;
633     }
635     FREE(del_element->name);
        FREE(del_element);
637 }
639 return 0;
641 }
643 /*****
644 *
645 * END OF FILE
646 *
647 *****/
```

A.3.3. Parallel Plug-in for Integral Computation

A.3.3.1. Header File

```
1 /* libintegral/integral.h. Generated by configure. */
   /*****
3  *
   * Header file for the Integral module.
5  * Copyright (c) Ronald Baumann
   *
7  * For more information see the following files in the source distribution top-
   * level directory or package data directory (usually /usr/local/share/package):
9  *
   * - README    for general package information.
11 * - INSTALL   for package install information.
   * - COPYING   for package license information and copying conditions.
13 * - AUTHORS   for package authors information.
   * - ChangeLog for package changes information.
15 *
   * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
17 * top-level directory to create the target file.
   *
19 *****/
21 /** \file integral.h
   * \brief Header file for integral module.
23 *
```

A. Appendix

```
25 * The libintegral module is a parallel plug-in for the Harness project. It
26 * calculates the integral of a function by using the Monte Carlo approach.
27 */
28 /* Avoid to include the content of this header file twice. */
29 #ifndef INTEGRAL_INTEGRAL_H
30 #define INTEGRAL_INTEGRAL_H
31
32 /*****
33 *
34 * Macros
35 *
36 *****/
37
38 /** \def LIBINTEGRAL_PACKAGE_NAME
39 * \brief Package name (unquoted).
40 */
41 #define LIBINTEGRAL_PACKAGE_NAME montecarlo
42
43
44 /** \def LIBINTEGRAL_PACKAGE_VERSION
45 * \brief Package version (unquoted).
46 */
47 #define LIBINTEGRAL_PACKAGE_VERSION 1.1
48
49
50 /** \def LIBINTEGRAL_PACKAGE_RELEASE
51 * \brief Package release (unquoted).
52 */
53 #define LIBINTEGRAL_PACKAGE_RELEASE 0
54
55
56 /** \def LIBINTEGRAL_PACKAGE_BUGREPORT
57 * \brief Package bug report e-mail (unquoted).
58 */
59 #define LIBINTEGRAL_PACKAGE_BUGREPORT baumannr@ornl.gov
60
61
62 /** \def LIBINTEGRAL_VERSION_CURRENT
63 * \brief Version current (unquoted).
64 */
65 #define LIBINTEGRAL_VERSION_CURRENT 0
66
67
68 /** \def LIBINTEGRAL_VERSION_REVISION
69 * \brief Version revision (unquoted).
70 */
71 #define LIBINTEGRAL_VERSION_REVISION 0
72
73
74 /** \def LIBINTEGRAL_VERSION_AGE
75 * \brief Version age (unquoted).
76 */
77 #define LIBINTEGRAL_VERSION_AGE 0
78
79
80 /** \def LIBINTEGRAL_VERSION_FIRST
81 * \brief Version first (unquoted).
82 */
83 #define LIBINTEGRAL_VERSION_FIRST 0
84
85
```

A. Appendix

```
87 /** \def LIBINTEGRAL_VERSION
   * \brief Version (unquoted).
89 */
#define LIBINTEGRAL_VERSION 0.0.0
91

93 /** \def INTEGRAL_QUOTES(string)
   * \brief Quoting a string.
95 */
#define INTEGRAL_QUOTES(string) #string
97

99 /** \def INTEGRAL_STRING(string)
   * \brief A string.
101 */
#define INTEGRAL_STRING(string) INTEGRAL_QUOTES(string)
103

105 /** \def INTEGRAL_FUSE(arg1, arg2)
   * \brief Fuse two strings.
107 */
#define INTEGRAL_FUSE(arg1, arg2) arg1##arg2
109

111 /** \def INTEGRAL_JOIN(arg1, arg2)
   * \brief Joining two text constants.
113 */
#define INTEGRAL_JOIN(arg1, arg2) INTEGRAL_FUSE(arg1, arg2)
115

117 /** \def INTEGRAL_WARN(string)
   * \brief Debug printout.
119 */
#define INTEGRAL_WARN(string) \
121     fprintf(stderr, \
122             INTEGRAL_STRING(warn: libintegral:%d:%s:%u:%s: string\n), \
123             getpid(), __FILE__, __LINE__, __FUNC__
125

127 /** \def INTEGRAL_INFO(string)
   * \brief Debug printout.
129 */
#define INTEGRAL_INFO(string) \
131     fprintf(stderr, \
132             INTEGRAL_STRING(info: libintegral:%d:%s:%u:%s: string\n), \
133             getpid(), __FILE__, __LINE__, __FUNC__
135

137 /** \def INTEGRAL_PRINT(string)
   * \brief Wrapper for debug printout.
139 */
#ifdef DEBUG
#define INTEGRAL_PRINT(string) string);
#else /* DEBUG */
141 #define INTEGRAL_PRINT(string)
142 #endif /* DEBUG */
143

145 /* Flag for <harness.0/harness.h> header. */
#ifndef HARNESS_HARNESS_H
147 #define HARNESS_HARNESS_H
148 #define HAVE_HARNESS_0_HARNESS_H 1
149 #endif /* HARNESS_0_HARNESS_H */
```

A. Appendix

```
#endif /* HARNESS_HARNESS_H */
151

153 /** \def FREE(x)
    * \brief Macro frees allocated memory and add the NULL pointer.
155 *
    * A NULL pointer is added after freeing allocated memory.
157 */
#define FREE(x) {if (x != NULL) {free(x); x = NULL;}}
159

161 /** \def TESTFILE
    * \brief Testfile for running the integration library without communication.
163 */
#define TESTFILE "/home/ronald/parallelplugins/data/montecarlo.input"
165

167 /** \def TESTMODE
    * \brief 0 = normal parallel plug-in mode, 1 = testmode without communication
169 * by using the testfile.
    */
171 #define TESTMODE 0

173
175 /* *****
    * Includes
177 *
    * *****/
179
    /* Include <harness.0/harness.h> header. */
181 #if HAVE_HARNESS_0_HARNESS_H
    #include <harness.0/harness.h>
183 #endif /* HAVE_HARNESS_0_HARNESS_H */

185 #include <math.h>
    #include "rmixintegral.h"
187

189 /* *****
    * Data Types
    * *****/
191
193
195 /** \struct integral_t
    * \brief Integral plug-in "class".
197 *
    * This structure contains the public plug-in data and function pointers.
199 * It also provides information about the version of the plug-in.
    */
201 typedef struct
    {
203     struct
        {
205         const unsigned int current; /* current version */
207         const unsigned int revision; /* current revision */
209         const unsigned int age; /* version age */
211         const unsigned int first; /* first supported version */
        } version; /* plug-in version */
    } integral_t;
```

A. Appendix

```
213 /*****
    *
215 * Function Prototypes
    *
217 *****/

219 /** \fn    int integral_init_rmix();
    * \brief  Initializes the necessary rmix parameter and exports the plug-in
221 *          communication interface.
    *
223 * \return 0 on success or -1 on any error
    */
225 int integral_init_rmix();

227
229 /** \fn    int integral_fini_rmix();
    * \brief  Finalizes rmix and unexports the plug-in.
    *
231 * \return 0 on success or -1 on any error
    */
233 int integral_fini_rmix();

235
237 /** \fn    RMIX_METHOD_CALL(rmixintegral_integration_call);
    * \brief  Invokes the integration (server-side method stub).
    *
239 * \param  object  The local object.
    * \param  outary  The output values array. The values array and its
241 *                containing values are allocated before the call with the
    *                exception of variable arrays and strings. They are allocated
243 *                dynamically or explicitly set to NULL by this function. In
    *                the case of variable arrays, the length value is allocated
245 *                before the call and is set to 0 if its variable array is
    *                NULL.
247 * \param  outcnt  The output values count.
    * \param  inary  The input values array. The values array and its containing
249 *                values are not modified or deallocated by this call.
    * \param  incnt  The input values count.
251 * \return  0 on success or -1 on any error with errno set
    *          appropriately.
    */
253 RMIX_METHOD_CALL(rmixintegral_integration_call);
255

257 /** \fn    int rmixintegral_integration ( double      *result, \
259                                     unsigned int  iterations, \
261                                     double        lowerbound, \
                                                double        upperbound, \
                                                unsigned int  numcoeffs, \
                                                double        *coeffs);
    * \brief  Calculates an integral with the received data and the Monte Carlo
263 *          approach.
    *
265 * \param  *result  Returns the calculated integral
267 * \param  iterations  Number of random Monte Carlo numbers
    * \param  lowerbound  Lower boundary of the integration interval
269 * \param  upperbound  Upper boundary of the integration interval
    * \param  numcoeffs  Number of coefficients
271 * \param  *coeffs  Array with coefficients
    * \return  0 on success or -1 on any error
273 */
275 int rmixintegral_integration ( double      *result,
                                unsigned int  iterations,
```

A. Appendix

```
277         double    lowerbound ,
278         double    upperbound ,
279         unsigned int numcoeffs ,
280         double    *coeffs );

281
282 /** \fn    double integral_func_value( unsigned int  num_coeffs, \
283                                     double        *coeffs, \
284                                     double        x_value);
285 * \brief  Calculates a function value of a specified function.
286 *
287 * \param  num_coeffs  Number of coefficients of the functions.
288 * \param  *coeffs    Array with the coefficients
289 * \param  x_value    Value, which is inserted into the function
290 * \return  The function value.
291 */
292 double integral_func_value( unsigned int  num_coeffs,
293                             double        *coeffs,
294                             double        x_value);
295
296
297 /** \fn    double integral_random_number( double bottom, \
298                                         double top);
299 * \brief  Generates a random number in a certain interval.
300 *
301 * \param  bottom  Lower boundary of the interval.
302 * \param  top    Upper boundary of the interval
303 * \return  The random number
304 */
305 double integral_random_number( double bottom,
306                               double top);
307
308
309 /*****
310 *
311 * The following section contains functions for testing the plug-in without
312 * the use of communication functions. A local input file is read and a
313 * integral is calculated.
314 *
315 *****/
316
317 /** \fn    int integral_calc_integral();
318 * \brief  Calculates the integral of a given function in a certain interval.
319 *
320 * \return 0 on success or -1 on any error
321 */
322 int integral_calc_integral();
323
324
325 /** \fn    int integral_read_inputfile( char        *filename, \
326                                       unsigned int *number, \
327                                       double       **coeffs, \
328                                       int          *iterations, \
329                                       double       *bottom, \
330                                       double       *top);
331 * \brief  Reads the input file, which contains the needed information.
332 *
333 * \param  *filename  Name of the input file
334 * \param  *number    Returns the number of coefficients
335 * \param  **coeffs   Returns the array of coefficients
336 * \param  *iterations Returns the amount of random numbers, which are used
337 * \param  *bottom    Returns the lower boundary of the interval
338 * \param  *top       Returns the upper boundary of the interval
```

A. Appendix

```
339 * \return          0 on success or -1 on any error
    */
341 int integral_read_inputfile( char      *filename ,
                               unsigned int *number ,
343                               double    **coeffs ,
                               int        *iterations ,
345                               double    *bottom ,
                               double    *top);
347
349 /*****
    *
351 * Data
    *
353 *****/
355 /** \var  extern integral_t integral
    * \brief Integral plug-in "object".
357 *
    * An external "object" of the Integral plug-in "class".
359 */
extern integral_t integral;
361
363 #endif /* INTEGRAL_INTEGRAL_H */
365
367 /*****
    *
369 * END OF FILE
    *
    *****/
```

A.3.3.2. Source File

```

/*****
2 *
    * Source file for the Integral module.
4 * Copyright (c) Ronald Baumann.
    *
6 * For more information see the following files in the source distribution top-
    * level directory or package data directory (usually /usr/local/share/package):
8 *
    * - README    for general package information.
10 * - INSTALL   for package install information.
    * - COPYING   for package license information and copying conditions.
12 * - AUTHORS   for package authors information.
    * - ChangeLog for package changes information.
14 *
    *****/
16
18 /** \file  integral.c
    * \brief Source file for integral module.
    *
20 * The libintegral module is a parallel plug-in for the Harness project. It
    * calculates the integral of a function by using the Monte Carlo approach.
22 */
24
26 /*****
    *
    * Includes
```

A. Appendix

```
28 *
    *****/
30 /* Main module header file. */
32 #include "integral.h"
34
36 *
38 * Data Types
    *****/
40
42 /** \struct integral_data_t
43 * \brief Integral plug-in module data type.
44 * Contains the mutex and the instances handler.
45 */
46 typedef struct
47 {
48     pthread_mutex_t mutex; /* mutex */
49     unsigned int count; /* instances array count */
50     struct
51     {
52         unsigned int handle; /* instance handle */
53     } *instances; /* instances array */
54 } integral_data_t;
56
58 /** \struct integral_rmix_data_t
59 * \brief Data for rmix communication functions.
60 */
61 typedef struct
62 {
63     rmix_localref_t *localref_harness; /* exported Harness kernel */
64     rmix_localref_t *localref_integral; /* exported integral object */
65     unsigned int handle_rmix; /* handle for rmix plug-in */
66 } integral_rmix_data_t;
68
70 * Function Prototypes
    *****/
72
74 /** \fn HARNESS_PLUGINS_INIT(integral_init);
75 * \brief Initializes the integral plug-in.
76 *
77 * \param handle The plug-in instance handle.
78 * \return 0 on success or -1 on any error with errno set appropriately.
79 */
80 HARNESS_PLUGINS_INIT(integral_init);
82
84 /** \fn HARNESS_PLUGINS_FINI(integral_fini);
85 * \brief Finalizes the integral plug-in.
86 *
87 * \param handle The plug-in instance handle.
88 * \return 0 on success or -1 on any error with errno set appropriately.
89 */
90 HARNESS_PLUGINS_FINI(integral_fini);
```


A. Appendix

```
92  /******
   *
94  * Data
   *
96  *****/

98  /** \var  const rmix_method_t rmixintegral_methods[RMIXINTEGRAL_METHODS_COUNT]
   * \brief Server-side method descriptors for rmix integral.
100 */
102 const rmix_method_t rmixintegral_methods[RMIXINTEGRAL_METHODS_COUNT] =
    RMIXINTEGRAL_METHODS;

104
106 /** \var  const rmix_interface_t rmixintegral_interface
   * \brief Server-side interface for rmix integral.
   */
108 const rmix_interface_t rmixintegral_interface =
    {
110     RMIXINTEGRAL_METHODS_COUNT,          /* method descriptor count */
        (rmix_method_t*)rmixintegral_methods /* method descriptor array */
112 };

114
116 /** \var  integral_data_t integral_data
   * \brief Integral plug-in module data.
   *
118 * Includes the mutex for instances handle and the instances array where the
   * handles are stored.
120 */
    integral_data_t integral_data =
122 {
124     PTHREAD_MUTEX_INITIALIZER, /* mutex */
        0, /* instances array count */
        NULL /* instances array */
126 };

128
130 /** \var  integral_rmix_data_t integral_rmix_data
   * \brief Data for rmix communication functions.
   *
132 * Includes the local references to the exported objects.
   */
134 integral_rmix_data_t integral_rmix_data =
    {
136     NULL, /* exported Harness kernel */
        NULL, /* exported integral object */
138     0 /* handle for rmix plug-in */
    };

140
142 /** \var  integral_t integral
   * \brief HIntegral plug-in "object".
144 *
   * Contains the version of the library and possible public data and function
146 * pointers.
   */
148 integral_t integral =
    {
150     {
152         LIBINTEGRAL_VERSION_CURRENT, /* current version */
            LIBINTEGRAL_VERSION_REVISION, /* current revision */
            LIBINTEGRAL_VERSION_AGE, /* version age */
        }
    }
};
```

A. Appendix

```
154     LIBINTEGRAL_VERSION_FIRST      /* first supported version */
155     }                               /* plug-in version          */
156 };

158
159 /******
160 *
161 * Functions
162 *
163 *****/
164
165 /*
166 * Initializes the Integral plug-in.
167 *
168 * handle = The plug-in instance handle.
169 * return = 0 on success or -1 on any error with errno set appropriately.
170 */
171 #undef __FUNC__
172 #define __FUNC__ "integral_init"
173 HARNESS_PLUGINS_INIT(integral_init)
174 {
175     int          error;
176     void         *instances;
177     unsigned int  index;
178
179     INTEGRAL_PRINT((
180         INTEGRAL_INFO(libintegral is starting)))
181
182     /* Lock integral plug-in mutex. */
183     if (0 != (error = pthread_mutex_lock(&integral_data.mutex)))
184     {
185         errno = error;
186         INTEGRAL_PRINT((
187             INTEGRAL_WARN(unable to lock integral plug-in mutex)))
188         harness_syserr();
189         return -1;
190     }
191
192     /* Search for handle in instances array. */
193     for (index = 0; index < integral_data.count; index++)
194     {
195         if (integral_data.instances[index].handle == handle)
196         {
197             INTEGRAL_PRINT((
198                 INTEGRAL_WARN(handle is already in instances array)))
199             harness_syserr();
200
201             /* Unlock integral plug-in mutex. */
202             if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
203             {
204                 errno = error;
205                 INTEGRAL_PRINT((
206                     INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
207                 harness_syserr();
208             }
209
210             return -1;
211         }
212     }
213
214     /* Increase instances array. */
215     index = integral_data.count;
216     integral_data.count++;

```

A. Appendix

```
218 /* Reallocate instances array. */
219 if (NULL == (instances = realloc( integral_data.instances ,
220                                 integral_data.count *
221                                 sizeof( integral_data.instances[0])))
222 {
223     INTEGRAL_PRINT((
224         INTEGRAL_WARN(unable to reallocate instances array)))
225     /* Unlock integral plug-in mutex. */
226     if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
227     {
228         errno = error;
229         INTEGRAL_PRINT((
230             INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
231         harness_syserr();
232     }
233     return -1;
234 }
235
236 integral_data.instances = instances;
237 /* Save instances array entry. */
238 integral_data.instances[index].handle = handle;
239 /* Check for first initialization. */
240 if (1 == integral_data.count)
241 {
242     /******
243     /* PUT YOUR INIT CODE HERE */
244     /******
245
246     /* test mode is selected, the plug-in will read in an input file
247     and compute an integral, otherwise it is part of the parallel
248     plug-in */
249     if (!TESTMODE)
250     {
251         /* if the communication interface cannot be exported terminate
252         the plug-in */
253         if (integral_init_rmix() != 0)
254         {
255             if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
256             {
257                 errno = error;
258                 INTEGRAL_PRINT((
259                     INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
260                 harness_syserr();
261                 return -1;
262             }
263             harness_kernel_shutdown();
264         }
265     }
266     /* For testing the plug-in functions without using rmix functions */
267     else
268     {
269         INTEGRAL_PRINT((
270             INTEGRAL_INFO(running in test mode)))
271         integral_calc_integral();
272         harness_kernel_shutdown();
273     }
274 }
275
276 /* Unlock integral plug-in mutex. */
277 if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
278 {
279     errno = error;
280     INTEGRAL_PRINT((
```

A. Appendix

```
280         INTEGRAL_WARN(unable to unlock integral plug-in mutex))
281         harness_syserr();
282         return -1;
283     }
284     return 0;
285 }
286
287
288 /*
289  * Finalizes the integral plug-in.
290  *
291  * handle = The plug-in instance handle.
292  * return = 0 on success or -1 on any error with errno set appropriately.
293  */
294 #undef __FUNC__
295 #define __FUNC__ "integral_fini"
296 HARNESS_PLUGINS_FINI(integral_fini)
297 {
298     int error;
299     void *instances;
300     unsigned int index;
301
302     INTEGRAL_PRINT((
303         INTEGRAL_INFO(libintegral is shutting down)))
304
305     /* Lock plug-in plug-in mutex. */
306     if (0 != (error = pthread_mutex_lock(&integral_data.mutex)))
307     {
308         errno = error;
309         INTEGRAL_PRINT((
310             INTEGRAL_WARN(unable to lock integral plug-in mutex)))
311         harness_syserr();
312         return -1;
313     }
314
315     /* Search for handle in instances array. */
316     for (index = 0; index < integral_data.count; index++)
317     {
318         if (integral_data.instances[index].handle == handle)
319         {
320             break;
321         }
322     }
323     /* Check if handle is not in instances array. */
324     if (index == integral_data.count)
325     {
326         INTEGRAL_PRINT((
327             INTEGRAL_WARN(handle is not in instances array)))
328         harness_syserr();
329         /* Unlock harness-example plug-in mutex. */
330         if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
331         {
332             errno = error;
333             INTEGRAL_PRINT((
334                 INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
335             harness_syserr();
336         }
337         return -1;
338     }
339
340     /* Remove instance from instances array. */
341     integral_data.count--;
342     memmove(integral_data.instances + index,
```

A. Appendix

```

    integral_data.instances + index +1,
344     (integral_data.count - index) *
        sizeof(unsigned int));
346
    /* Reallocate instances array. */
348     if (0 == integral_data.count)
        {
350         free(integral_data.instances);
            integral_data.instances = NULL;
352     }
    else if (NULL == (instances = realloc( integral_data.instances ,
354                                         integral_data.count *
                                            sizeof( integral_data.instances[0]))))
356     {
        INTEGRAL_PRINT((
358             INTEGRAL_WARN(unable to reallocate instances array)))
    }
360     else
        {
362         integral_data.instances = instances;
        }
364     /* Check for last finalization. */
    if (0 == integral_data.count)
366     {
        /******
368         /* PUT YOUR FINI CODE HERE */
        /******
370         integral_fini_rmix();
    }
372
    /* Unlock integral plug-in mutex. */
374     if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
        {
376         errno = error;
            INTEGRAL_PRINT((
378             INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
            harness_syserr();
380         return -1;
        }
382     return 0;
}
384

386 /* Initializes the necessary rmix parameter and exports the plug-ins.
    *
388 * \return 0 on success or -1 on any error
    */
390 #undef __FUNC__
#define __FUNC__ "integral_init_rmix"
392 int integral_init_rmix()
    {
394     int ret;

396     RMIX_LOG((RMIX_INFO(start initialization of RMIX)))
    INTEGRAL_PRINT((
398         INTEGRAL_INFO(start initialization of RMIX)))

400     /* Load RMIX plug-in and initialize RMIX */

402     ret = harness_plugins_load( &integral_rmix_data.handle_rmix ,
                                "libharness-rmix.0", HARNESS_PLUGINS_EXPORT);
404     if (ret != 0)
        {
```

A. Appendix

```
406     INTEGRAL_PRINT((
407         INTEGRAL_WARN(unable to load RMIX plug-in)))
408     return -1;
409 }
410
411 /* Exports Harness while forcing 1000 as specific object handle */
412 ret = harness_rmix_export4( &integral_rmix_data.localref_harness ,
413                             "PROTOCOL=RPC OBJECTID=1000");
414 if (ret != 0)
415 {
416     RMIX_LOG((RMIX_WARN(unable to export Harness kernel)))
417     return -1;
418 }
419
420 /* Exports integral plug-in while forcing 1001 as specific object handle */
421 ret = rmix_export4( &integral_rmix_data.localref_integral ,
422                    "PROTOCOL=RPC OBJECTID=1001", NULL, &rmixintegral_interface);
423 if (ret != 0)
424 {
425     RMIX_LOG((RMIX_WARN(unable to export integral interface)))
426     return -1;
427 }
428
429 INTEGRAL_PRINT((
430     INTEGRAL_INFO(RMIX initialized and integral plug-in exported)))
431
432 return 0;
433 }
434
435 /* Finalizes rmix and unexports the plug-ins.
436 *
437 * \return 0 on success or -1 on any error
438 */
439 #undef __FUNC__
440 #define __FUNC__ "integral_fini_rmix"
441 int integral_fini_rmix()
442 {
443     int ret;
444     int err = 0;
445
446     INTEGRAL_PRINT((
447         INTEGRAL_INFO(start finalization of RMIX)))
448
449     /* Unexports integral plug-in */
450     ret = rmix_unexport( &integral_rmix_data.localref_integral);
451     if (ret != 0)
452     {
453         RMIX_LOG((RMIX_WARN(unable to unexport integral interface)))
454         err = -1;
455     }
456
457     /* Unexports Harness */
458     ret = harness_rmix_unexport( &integral_rmix_data.localref_harness);
459     if (ret != 0)
460     {
461         RMIX_LOG((RMIX_WARN(unable to unexport Harness kernel)))
462         err = -1;
463     }
464
465     /* Unload RMIX plug-in */
466     ret = harness_plugins_unload( integral_rmix_data.handle_rmix);
467     if (ret != 0)
```

A. Appendix

```
470     {
471         INTEGRAL_PRINT((
472             INTEGRAL_WARN(unable to unload RMX plug-in)))
473     }
474
475     INTEGRAL_PRINT((
476         INTEGRAL_INFO(RMX finalized and integral plug-in unexported)))
477
478     return err;
479 }
480
481 /*
482  * Invokes the integration (server-side method stub).
483  *
484  * Calculates an integral with the received data and the Monte Carlo approach.
485  *
486  * \param object The local object.
487  * \param outary The output values array. The values array and its
488  *               containing values are allocated before the call with the
489  *               exception of variable arrays and strings. They are
490  *               allocated dynamically or explicitly set to NULL by this
491  *               function. In the case of variable arrays, the length value
492  *               is allocated before the call and is set to 0 if its
493  *               variable array is NULL.
494  * \param outcnt The output values count.
495  * \param inary The input values array. The values array and its containing
496  *              values are not modified or deallocated by this call.
497  * \param incnt The input values count.
498  * \return      0 on success or -1 on any error with errno set
499  *              appropriately.
500  */
501 #undef __FUNC__
502 #define __FUNC__ "rmxintegral_integration_call"
503 RMX_METHOD_CALL(rmxintegral_integration_call)
504 {
505     int result;
506
507     unsigned int iterations;
508     double lowerbound;
509     double upperbound;
510     unsigned int numcoeffs;
511     double *coeffs;
512     double integral;
513
514 #ifdef DEBUG
515     /* Check object parameter. */
516     /*
517     if (NULL == object)
518     {
519         errno = EINVAL;
520         RMX_LOG((RMX_WARN(object parameter is null)))
521         errno = EINVAL;
522         return -1;
523     }
524     */
525
526     /* Check outary parameter. */
527     if ((NULL == outary)&&(0 != outcnt))
528     {
529         errno = EINVAL;
530         RMX_LOG((RMX_WARN(outary parameter is null)))

```

A. Appendix

```
532     errno = EINVAL;
533     return -1;
534 }
535
536 /* Check outcnt parameter. */
537 if ((0 == outcnt)&&(NULL != outary))
538 {
539     errno = EINVAL;
540     RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
541     errno = EINVAL;
542     return -1;
543 }
544
545 /* Check inary parameter. */
546 if ((NULL == inary)&&(0 != incnt))
547 {
548     errno = EINVAL;
549     RMIX_LOG((RMIX_WARN(inary parameter is null)))
550     errno = EINVAL;
551     return -1;
552 }
553
554 /* Check incnt parameter. */
555 if ((0 == incnt)&&(NULL != inary))
556 {
557     errno = EINVAL;
558     RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
559     errno = EINVAL;
560     return -1;
561 }
562 #endif /* DEBUG */
563
564 /* Prepare input. */
565 iterations = *(unsigned int*)inary[0];
566 lowerbound = *(double*)    inary[1];
567 upperbound = *(double*)    inary[2];
568 numcoeffs  = *(unsigned int*)inary[3];
569 coeffs     = (double*)    inary[4];
570
571 /* Call method function. */
572 result = rmixintegral_integration(&integral, iterations, lowerbound,
573                                 upperbound, numcoeffs, coeffs);
574
575 /* Prepare output. */
576 *(int*)  (outary[0]) = result;
577 *(double*)(outary[1]) = integral;
578
579 /* Reset errno if needed. */
580 if (0 != result)
581 {
582     errno = 0;
583 }
584 return 0;
585 }
586
587 /*
588 * Calculates an integral with the received data and the Monte Carlo approach.
589 *
590 * \param *result Returns the calculated integral
591 * \param iterations Number of random Monte Carlo numbers
592 * \param lowerbound Lower boundary of the integration interval
593 * \param upperbound Upper boundary of the integration interval
594 */
```


A. Appendix

```

    * \param    numcoeffs    Number of coefficients
596 * \param    *coeffs      Array with coefficients
    * \return          0 on success or -1 on any error
598 */
#undef __FUNC__
600 #define __FUNC__ "rmixintegral_integration"
int rmixintegral_integration ( double      *result ,
602                          unsigned int  iterations ,
                          double      lowerbound ,
604                          double      upperbound ,
                          unsigned int  numcoeffs ,
606                          double      *coeffs )
{
608     int i;
#ifdef DEBUG
610     /* check array parameter */
    if (coeffs == NULL)
612     {
        INTEGRAL_PRINT((
614             INTEGRAL_WARN(array of coefficients contains no data)))
        return -1;
616     }
#endif /* DEBUG */
618     /* initialise the random generator */
620     srand(time(NULL));

622     /* computation */
    for ( i=0; i<iterations; i++)
624     {
        (*result) += integral_func_value( numcoeffs, coeffs ,
626                                     integral_random_number( lowerbound, upperbound));
    }
628     (*result) = (*result) * (upperbound - lowerbound) / iterations;

630     /* for testing purposes, the plug-in was slowed so that it can be shutdown
        manually */
632     /*
634     RMIX_LOG((RMIX_INFO(----- )))
        RMIX_LOG((RMIX_INFO(beende plugin lege plugin schalfen )))
        RMIX_LOG((RMIX_INFO(----- )))
636     sleep(30);
        */
638     INTEGRAL_PRINT((
640             INTEGRAL_INFO(integral is %lf), (*result)))

642     return 0;
644 }

646 /*
    * Calculates the function value of a given function.
648 *
    * \param    num_coeffs    Number of coefficients of the functions.
650 * \param    *coeffs      Array with the coefficients
    * \param    x_value      Value, which is inserted into the function
652 * \return          The function value.
    */
#undef __FUNC__
#define __FUNC__ "integral_func_value"
656 double integral_func_value( unsigned int  num_coeffs ,
                          double      *coeffs ,
```

A. Appendix

```
658         double      x_value)
659     {
660     int i;
661     double erg = 0;
662     for ( i=0; i<num_coefs; i++)
663     {
664         erg += coefs[i] * pow(x_value, (double) i);
665     }
666     return erg;
667 }
668
669 /*
670 * Generates a random number in a certain interval.
671 *
672 * \param bottom Lower boundary of the interval.
673 * \param top    Upper boundary of the interval
674 * \return      The random number
675 */
676 #undef __FUNC__
677 #define __FUNC__ "integral_random_number"
678 double integral_random_number( double bottom,
679                               double top)
680 {
681     double random = 0;
682     random = (double)rand() / RAND_MAX;
683     random = random * (top - bottom) + bottom;
684     return random;
685 }
686
687
688 /******
689 *
690 * The following section contains for testing the plug-in without the use
691 * of communication functions. A local input file is read and a integral
692 * is calculated.
693 *
694 *
695 */
696
697 /*
698 * Calculates the integral of a given function in a certain interval.
699 *
700 * \return 0 on success or -1 on any error
701 */
702 #undef __FUNC__
703 #define __FUNC__ "integral_calc_integral"
704 int integral_calc_integral()
705 {
706     char *filename;
707
708     unsigned int num_of_coefs; /* number of coefficients */
709     int iterations; /* number of iterations */
710     double *coefs; /* array for storing the coefficients */
711
712     double bottom_border; /* bottom border for the calculation area */
713     double top_border; /* top border of the calculation area */
714
715     double result = 0; /* stores the result */
716     int i, ret;
717
718     /* example file for testing the plug-in without a loader */
719     filename = TESTFILE;
720
```

A. Appendix

```
722  /* read the input file */
    ret = integral_read_inputfile( filename, &num_of_coefs, &coefs,
                                &iterations, &bottom_border, &top_border);
724  if ( ret == -1 )
    {
726      INTEGRAL_PRINT((
          INTEGRAL_WARN(unable to read input file %s), filename))
728      return -1;
    }
730
    /* initialise the random generator */
732    srand(time(NULL));
734
    /* computation */
    for ( i=0; i<iterations; i++)
736    {
        result += integral_func_value( num_of_coefs, coefs,
738                                     integral_random_number( bottom_border, top_border));
    }
740    result = result * (top_border - bottom_border) / iterations;
742
    INTEGRAL_PRINT((
        INTEGRAL_INFO(integral is %lf), result))
744
    FREE(coefs);
746    return 0;
}
748
750 /*
 * Reads the input file, which contains the needed information.
752 *
 * \param *filename Name of the input file
754 * \param *number Returns the number of coefficients
 * \param **coefs Returns the array of coefficients
756 * \param *iterations Returns the amount of random numbers, which are used
 * \param *bottom Returns the lower boundary of the interval
758 * \param *top Returns the upper boundary of the interval
 * \return 0 on success or -1 on any error
760 */
#undef __FUNC__
762 #define __FUNC__ "integral_read_inputfile"
int integral_read_inputfile( char *filename,
764                          unsigned int *number,
                          double **coefs,
766                          int *iterations,
                          double *bottom,
768                          double *top)
{
770     FILE *dat_file; /* input file */
    int i;
772     int ret;
774
    /* open the file with the coefficients */
776     dat_file = fopen(filename, "r");
    if (dat_file == NULL)
778     {
        INTEGRAL_PRINT((
780             INTEGRAL_WARN(unable to open config file)))
        return -1;
    }
782     else
```

A. Appendix

```
784 {
785     INTEGRAL_PRINT((
786         INTEGRAL_INFO(input file is opened)))
787 }
788
789 /* read in the number of iterations */
790 ret = fscanf(dat_file, "%d", iterations);
791 if (ret == 0 || ret == -1)
792 {
793     INTEGRAL_PRINT((
794         INTEGRAL_WARN(unable to read iterations from file)))
795     fclose(dat_file);
796     return -1;
797 }
798
799 /* read in the boundaries */
800 ret = fscanf(dat_file, "%lf", bottom);
801 if (ret == 0 || ret == -1)
802 {
803     INTEGRAL_PRINT((
804         INTEGRAL_WARN(unable to read lower boundary from file)))
805     fclose(dat_file);
806     return -1;
807 }
808 ret = fscanf(dat_file, "%lf", top);
809 if (ret == 0 || ret == -1)
810 {
811     INTEGRAL_PRINT((
812         INTEGRAL_WARN(unable to read upper boundary from file)))
813     fclose(dat_file);
814     return -1;
815 }
816
817 /* read in the number of coefficients */
818 ret = fscanf(dat_file, "%d", number);
819 if (ret == 0 || ret == -1)
820 {
821     INTEGRAL_PRINT((
822         INTEGRAL_WARN(unable to read number of coefficients from
823             file)))
824     fclose(dat_file);
825     return -1;
826 }
827
828 /* create the array */
829 (*coeffs) = (double *)malloc(sizeof(double) * (*number));
830 if ((*coeffs) == NULL)
831 {
832     INTEGRAL_PRINT((
833         INTEGRAL_WARN(unable to allocate memory for coefficients)))
834     fclose(dat_file);
835     return -1;
836 }
837
838 /* read in the coefficients */
839 for (i=0; i<(*number); i++)
840 {
841     ret = fscanf(dat_file, "%lf", &(*coeffs)[i]);
842     if (ret == 0 || ret == -1)
843     {
844         INTEGRAL_PRINT((
845             INTEGRAL_WARN(unable to read coefficients from file)))
846         FREE(coeffs);
```

A. Appendix

```
848         fclose(dat_file);
            return -1;
850     }
852     fclose(dat_file);
        return 0;
854 }

856 /*****
858 *
860 *****/
```

A.3.4. Parallel Plug-in for a Image Processing Pipeline

A.3.4.1. Header File

```
/* libimgproc/imgproc.h.  Generated by configure. */
2 /*****
   *
   4 * Header file for the image processing module.
   * Copyright (c) Ronald Baumann
   6 *
   * For more information see the following files in the source distribution top-
   8 * level directory or package data directory (usually /usr/local/share/package):
   *
  10 * - README    for general package information.
   * - INSTALL  for package install information.
  12 * - COPYING  for package license information and copying conditions.
   * - AUTHORS  for package authors information.
  14 * - ChangeLog for package changes information.
   *
  16 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
   * top-level directory to create the target file.
  18 *
   *****/
20
  22 /** \file imgproc.h
   * \brief Header file for image processing module.
   *
  24 * The libimgproc module is a plug-in for the Harness project. It performs
   * various image filters on pictures.
  26 */

  28 /* Avoid to include the content of this header file twice. */
   #ifndef IMGPROC_IMGPROC_H
  30 #define IMGPROC_IMGPROC_H

  32
   /*****
  34 *
   * Macros
  36 *
   *****/
  38
   /** \def  LIBIMGPROC_PACKAGE_NAME
  40 * \brief Package name (unquoted).
   */
```

A. Appendix

```
42 #define LIBIMGPROC_PACKAGE_NAME imageprocessing
44
45 /** \def LIBIMGPROC_PACKAGE_VERSION
46 * \brief Package version (unquoted).
47 */
48 #define LIBIMGPROC_PACKAGE_VERSION 1.1
49
50 /** \def LIBIMGPROC_PACKAGE_RELEASE
51 * \brief Package release (unquoted).
52 */
53 #define LIBIMGPROC_PACKAGE_RELEASE 0
54
55 /** \def LIBIMGPROC_PACKAGE_BUGREPORT
56 * \brief Package bug report e-mail (unquoted).
57 */
58 #define LIBIMGPROC_PACKAGE_BUGREPORT baumannr@ornl.gov
59
60 /** \def LIBIMGPROC_VERSION_CURRENT
61 * \brief Version current (unquoted).
62 */
63 #define LIBIMGPROC_VERSION_CURRENT 0
64
65 /** \def LIBIMGPROC_VERSION_REVISION
66 * \brief Version revision (unquoted).
67 */
68 #define LIBIMGPROC_VERSION_REVISION 0
69
70 /** \def LIBIMGPROC_VERSION_AGE
71 * \brief Version age (unquoted).
72 */
73 #define LIBIMGPROC_VERSION_AGE 0
74
75 /** \def LIBIMGPROC_VERSION_FIRST
76 * \brief Version first (unquoted).
77 */
78 #define LIBIMGPROC_VERSION_FIRST 0
79
80 /** \def LIBIMGPROC_VERSION
81 * \brief Version (unquoted).
82 */
83 #define LIBIMGPROC_VERSION 0.0.0
84
85 /** \def IMGPROC_QUOTES(string)
86 * \brief Quoting a string.
87 */
88 #define IMGPROC_QUOTES(string) #string
89
90 /** \def IMGPROC_STRING(string)
91 * \brief A string.
92 */
93 #define IMGPROC_STRING(string) IMGPROC_QUOTES(string)
94
95
96
97
98
99
100
101
102
103
104
```

A. Appendix

```
106  /** \def IMGPROC_FUSE(arg1, arg2)
    * \brief Fuse two strings.
    */
108 #define IMGPROC_FUSE(arg1, arg2) arg1##arg2

110
111  /** \def IMGPROC_JOIN(arg1, arg2)
    * \brief Joining two text constants.
    */
114 #define IMGPROC_JOIN(arg1, arg2) IMGPROC_FUSE(arg1, arg2)

116
117  /** \def IMGPROC_WARN(string)
    * \brief Debug printout.
    */
120 #define IMGPROC_WARN(string) \
    fprintf(stderr, \
122         IMGPROC_STRING(warn: libimgproc:%d:%s:%u:%s: string\n), \
        getpid(), __FILE__, __LINE__, __FUNC__
124

126  /** \def IMGPROC_INFO(string)
    * \brief Debug printout.
    */
128 #define IMGPROC_INFO(string) \
130     fprintf(stderr, \
        IMGPROC_STRING(info: libimgproc:%d:%s:%u:%s: string\n), \
132         getpid(), __FILE__, __LINE__, __FUNC__

134
135  /** \def IMGPROC_PRINT(string)
    * \brief Wrapper for debug printout.
    */
138 #ifdef DEBUG
    #define IMGPROC_PRINT(string) string);
140 #else /* DEBUG */
    #define IMGPROC_PRINT(string)
142 #endif /* DEBUG */

144
145  /* Flag for <harness.0/harness.h> header. */
146 #ifndef HARNESS_HARNESS_H
    #define HARNESS_HARNESS_H
148 #define HAVE_HARNESS_0_HARNESS_H 1
    #endif /* HAVE_HARNESS_0_HARNESS_H */
150 #endif /* HARNESS_HARNESS_H */

152
153  /** \def FREE(x)
    * \brief Macro frees allocated memory and add the NULL pointer.
    *
    * A NULL pointer is added after freeing allocated memory.
    */
158 #define FREE(x) {if (x != NULL) {free(x); x = NULL;}}

160
161  /** \def ThrowWandException(wand)
    * \brief Prints out error information caused by Magick Wand library.
    */
164 #define ThrowWandException(wand) \
    { \
166     char *description; \
        ExceptionType severity; \
```

A. Appendix

```
168     description = MagickGetException( wand, &severity); \
169     fprintf( stderr, "%s %s %ld %s\n", GetMagickModule(), description); \
170     description = (char *) MagickRelinquishMemory(description); \
171 }
172
173
174 /** \def FILTER_NUMBER
175     * \brief Number of image filters.
176 */
177 #define FILTER_NUMBER 5
178
179
180 /*****
181 *
182 * Includes
183 *
184 *****/
185
186 /* Include <harness.0/harness.h> header. */
187 #if HAVE_HARNESS_0_HARNESS_H
188 #include <harness.0/harness.h>
189 #endif /* HAVE_HARNESS_0_HARNESS_H */
190
191 #include <wand/magick_wand.h>
192 #include <dirent.h>
193 #include "rmiximgproc.h"
194 #include "rmixppm.h"
195
196
197 /*****
198 *
199 * Data Types
200 *
201 *****/
202
203 /** \struct imgproc_t
204     * \brief Integral plug-in "class".
205     *
206     * This structure contains the public plug-in data and function pointers.
207     * It also provides information about the version of the plug-in.
208 */
209 typedef struct
210 {
211     struct
212     {
213         const unsigned int current; /* current version */
214         const unsigned int revision; /* current revision */
215         const unsigned int age; /* version age */
216         const unsigned int first; /* first supported version */
217     } version; /* plug-in version */
218 } imgproc_t;
219
220
221 /** \struct worklist_t
222     * \brief Element of the linked list for storing image information.
223     *
224     * Each element contains name, size and the image data from images, which were
225     * processed by the plug-in.
226 */
227 typedef struct worklist
228 {
229     char *name; /* name of the image */
230     size_t size; /* size of the image */
231 }
```


A. Appendix

```
232     unsigned char    *blob;        /* image data */
233     struct    worklist *next;      /* pointer to the next list element */
234 }worklist_t;

236 /*****
237 *
238 * Function Prototypes
239 *
240 *****/

242 /** \fn    int imgproc_init_rmix();
243 * \brief  Initializes the necessary rmix parameter and exports the plug-in.
244 *
245 * \return 0 on success or -1 on any error
246 */
247 int imgproc_init_rmix();

250 /** \fn    int imgproc_fini_rmix();
251 * \brief  Finalizes rmix and unexports the plug-in.
252 *
253 * \return 0 on success or -1 on any error
254 */
255 int imgproc_fini_rmix();

258 /** \fn    RMIX_METHOD_CALL(rmiximgproc_initpipeline_call);
259 * \brief  Accepts calls for the initialisation of the pipeline component
260 *         (server-side method stub).
261 *
262 * \param  object    The local object.
263 * \param  outary    The output values array. The values array and its
264 *                 containing values are allocated before the call with the
265 *                 exception of variable arrays and strings. They are allocated
266 *                 dynamically or explicitly set to NULL by this function. In
267 *                 the case of variable arrays, the length value is allocated
268 *                 before the call and is set to 0 if its variable array is
269 *                 NULL.
270 * \param  outcnt    The output values count.
271 * \param  inary     The input values array. The values array and its containing
272 *                 values are not modified or deallocated by this call.
273 * \param  incnt     The input values count.
274 * \return 0 on success or -1 on any error.
275 */
276 RMIX_METHOD_CALL(rmiximgproc_initpipeline_call);

278 /** \fn    int rmiximgproc_initpipeline (unsigned int    filter, \
279                                         char            *sourcedir, \
280                                         char            *targetdir, \
281                                         char            *successor \
282                                         char            *predecessor \
283                                         char            *ppmnode);
284 * \brief  Initialises the global pipeline parameters.
285 *
286 * \param  filter      Filter which will be used
287 * \param  *sourcedir  Source images dir
288 * \param  *targetdir  Target dir for images
289 * \param  *sucessor   Successor plug-in
290 * \param  *predecessor Predecessor plug-in
291 * \param  *ppmnode    Node running the PPM
292 * \return 0 on success or -1 on any error.
```

A. Appendix

```
294 */
int rmiximgproc_initpipeline (unsigned int filter ,
296 char *sourcedir ,
char *targetdir ,
298 char *successor ,
char *predecessor ,
300 char *ppmnode);

302
/** \fn RMIX_METHOD_CALL(rmiximgproc_invokepipeline_call);
304 * \brief Accepts invocation calls for the pipeline (server-side method stub).
*
306 * \param object The local object.
* \param outary The output values array. The values array and its
308 * containing values are allocated before the call with the
* exception of variable arrays and strings. They are allocated
310 * dynamically or explicitly set to NULL by this function. In
* the case of variable arrays, the length value is allocated
312 * before the call and is set to 0 if its variable array is
* NULL.
314 * \param outcnt The output values count.
* \param inary The input values array. The values array and its containing
316 * values are not modified or deallocated by this call.
* \param incnt The input values count.
318 * \return 0 on success or -1 on any error.
*/
320 RMIX_METHOD_CALL(rmiximgproc_invokepipeline_call);

322
/** \fn int rmiximgproc_invokepipeline ();
324 * \brief Starts the image processing pipeline.
*
326 * \return 0 on success or -1 on any error.
*/
328 int rmiximgproc_invokepipeline ();

330
/** \fn RMIX_METHOD_CALL(rmiximgproc_availabilitycheck_call);
332 * \brief Accepts checking calls, if the plug-in was loaded correctly
* (server-side method stub).
334 *
* \param object The local object.
336 * \param outary The output values array. The values array and its
* containing values are allocated before the call with the
338 * exception of variable arrays and strings. They are allocated
* dynamically or explicitly set to NULL by this function. In
340 * the case of variable arrays, the length value is allocated
* before the call and is set to 0 if its variable array is
342 * NULL.
* \param outcnt The output values count.
344 * \param inary The input values array. The values array and its containing
* values are not modified or deallocated by this call.
346 * \param incnt The input values count.
* \return 0 on success or -1 on any error.
348 */
RMIX_METHOD_CALL(rmiximgproc_availabilitycheck_call);
350

352 /** \fn RMIX_METHOD_CALL(rmiximgproc_setimagecounter_call);
* \brief Accepts initialisation calls for the image counter of the plug-in
354 * (server-side method stub).
*
356 * \param object The local object.
```

A. Appendix

```

358 * \param outary The output values array. The values array and its
* containing values are allocated before the call with the
* exception of variable arrays and strings. They are allocated
360 * dynamically or explicitly set to NULL by this function. In
* the case of variable arrays, the length value is allocated
362 * before the call and is set to 0 if its variable array is
* NULL.
364 * \param outcnt The output values count.
* \param inary The input values array. The values array and its containing
366 * values are not modified or deallocated by this call.
* \param incnt The input values count.
368 * \return 0 on success or -1 on any error.
*/
370 RMX_METHOD_CALL(rmiximgproc_setimagecounter_call);

372
/** \fn int rmiximgproc_setimagecounter( unsigned int counter);
374 * \brief Sets the image counter of the plug-in.
*
376 * \param counter Number of images to process
* \return 0 on success or -1 on any error.
378 */
int rmiximgproc_setimagecounter( unsigned int counter);
380

382 /** /fn int rmiximgprocclient_setimagecounter( rmix_remoteref_t *remoteobj,\
unsigned int counter);
384 * \brief Calls the function for setting an image counter at another plug-in
* (client-side method stub).
386 *
* \param *remoteobj ID of the remote object
388 * \param counter Image counter
* \return 0 on success or -1 on any error
390 */
int rmiximgprocclient_setimagecounter ( rmix_remoteref_t *remoteobj ,
392 unsigned int counter);

394
/** \fn RMX_METHOD_CALL(rmiximgproc_passimage_call);
396 * \brief Accepts callsm, which forward an image to the plug-in
* (server-side method stub).
398 *
* \param object The local object.
400 * \param outary The output values array. The values array and its
* containing values are allocated before the call with the
402 * exception of variable arrays and strings. They are allocated
* dynamically or explicitly set to NULL by this function. In
404 * the case of variable arrays, the length value is allocated
* before the call and is set to 0 if its variable array is
406 * NULL.
* \param outcnt The output values count.
408 * \param inary The input values array. The values array and its containing
* values are not modified or deallocated by this call.
410 * \param incnt The input values count.
* \return 0 on success or -1 on any error.
412 */
RMX_METHOD_CALL(rmiximgproc_passimage_call);
414

416 /** \fn int rmiximgproc_passimage (char *name, \
size_t size, \
418 unsigned char *blob);
* \brief Processes and then forwards an image to the next plug-in.
```

A. Appendix

```
420 *
421 * \param size Size of image data
422 * \param *blob Image data
423 * \return 0 on success or -1 on any error.
424 */
425 int rmiximgproc_passimage (char *name,
426 size_t size,
427 unsigned char *blob);
428
429
430 /** \fn int rmiximgprocclient_passimage_oneway (
431 rmix_remoteref_t *remoteobj, \
432 char *name,\
433 unsigned char *blob,\
434 unsigned int size);
435 * \brief Calls a plug-in for forwarding an image (client-side method stub).
436 *
437 * \param *name Name of the image file
438 * \param *blob Image data
439 * \param size Size of the image data
440 * \return 0 on success or -1 on any error
441 */
442 int rmiximgprocclient_passimage_oneway ( rmix_remoteref_t *remoteobj,
443 char *name,
444 unsigned char *blob,
445 unsigned int size);
446
447
448 /** \fn RMIX_METHOD_CALL(rmiximgproc_updatesuccessor_call);
449 * \brief Accepts calls for updating the predecessor entry of the plug-in.
450 * (server-side stub)
451 *
452 * \param object The local object.
453 * \param outary The output values array. The values array and its
454 * containing values are allocated before the call with the
455 * exception of variable arrays and strings. They are allocated
456 * dynamically or explicitly set to NULL by this function. In
457 * the case of variable arrays, the length value is allocated
458 * before the call and is set to 0 if its variable array is
459 * NULL.
460 * \param outcnt The output values count.
461 * \param inary The input values array. The values array and its containing
462 * values are not modified or deallocated by this call.
463 * \param incnt The input values count.
464 * \return 0 on success or -1 on any error.
465 */
466 RMIX_METHOD_CALL(rmiximgproc_updatesuccessor_call);
467
468
469 /** \fn int rmiximgproc_updatesuccessor (char *predecessor);
470 * \brief Updates the predecessor entry of the plug-in.
471 *
472 * \param *predecessor Name of the predecessor plug-in.
473 * \return 0 on success or -1 on any error.
474 */
475 int rmiximgproc_updatesuccessor (char *predecessor);
476
477
478 /** \fn RMIX_METHOD_CALL(rmiximgproc_updatepredecessor_call);
479 * \brief Accepts calls for updating the successor entry of the plug-in and
480 * also returns the current image counter. (server-side stub)
481 *
482 * \param object The local object.
```

A. Appendix

```
* \param outary The output values array. The values array and its
484 * containing values are allocated before the call with the
* exception of variable arrays and strings. They are allocated
486 * dynamically or explicitly set to NULL by this function. In
* the case of variable arrays, the length value is allocated
488 * before the call and is set to 0 if its variable array is
* NULL.
490 * \param outcnt The output values count.
* \param inary The input values array. The values array and its containing
492 * values are not modified or deallocated by this call.
* \param incnt The input values count.
494 * \return 0 on success or -1 on any error.
*/
496 RMIX_METHOD_CALL(rmiximgproc_updatepredecessor_call);

498
/** \fn int rmiximgproc_updatepredecessor( char *successor, \
500 unsigned int *counter);
* \brief Updates the successor entry of the plug-in and returns the current
502 * image counter.
*
504 * \param *successor Name of the successor plug-in.
* \param *counter Current image counter.
506 * \return 0 on success or -1 on any error.
*/
508 int rmiximgproc_updatepredecessor( char *successor,
unsigned int *counter);
510

512 /** \fn RMIX_METHOD_CALL(rmiximgproc_updateimagecounter_call);
* \brief Accepts calls for updating the image counter. (server-side stub)
514 *
* \param object The local object.
516 * \param outary The output values array. The values array and its
* containing values are allocated before the call with the
518 * exception of variable arrays and strings. They are allocated
* dynamically or explicitly set to NULL by this function. In
520 * the case of variable arrays, the length value is allocated
* before the call and is set to 0 if its variable array is
522 * NULL.
* \param outcnt The output values count.
524 * \param inary The input values array. The values array and its containing
* values are not modified or deallocated by this call.
526 * \param incnt The input values count.
* \return 0 on success or -1 on any error.
528 */
RMIX_METHOD_CALL(rmiximgproc_updateimagecounter_call);
530

532 /** \fn int rmiximgproc_updateimagecounter( unsigned int counter);
* \brief Updates the image counter.
534 *
* \param counter Number of images to process
536 * \return 0 on success or -1 on any error.
*/
538 int rmiximgproc_updateimagecounter( unsigned int counter);

540
/** \fn RMIX_METHOD_CALL(rmiximgproc_sendworklist_call);
542 * \brief Accepts call for sending the images in the work list again to the
* successor plug-in (server-side stub).
544 *
* \param object The local object.
```

A. Appendix

```
546 * \param outary The output values array. The values array and its
* containing values are allocated before the call with the
548 * exception of variable arrays and strings. They are allocated
* dynamically or explicitly set to NULL by this function. In
550 * the case of variable arrays, the length value is allocated
* before the call and is set to 0 if its variable array is
552 * NULL.
* \param outcnt The output values count.
554 * \param inary The input values array. The values array and its containing
* values are not modified or deallocated by this call.
556 * \param incnt The input values count.
* \return 0 on success or -1 on any error.
558 */
RMIX_METHOD_CALL(rmiximgproc_sendworklist_call);
560

562 /** \fn int rmiximgproc_sendworklist();
* \brief Sends again the images in the worklist to the successor plug-in.
564 *
* \return 0 on success or -1 on any error.
566 */
int rmiximgproc_sendworklist();
568

570 /** \fn RMIX_METHOD_CALL(rmiximgproc_imageprocessed_call);
* \brief Accepts notification calls that an image was successfully stored
572 * (server-side stub).
*
574 * \param object The local object.
* \param outary The output values array. The values array and its
576 * containing values are allocated before the call with the
* exception of variable arrays and strings. They are allocated
578 * dynamically or explicitly set to NULL by this function. In
* the case of variable arrays, the length value is allocated
580 * before the call and is set to 0 if its variable array is
* NULL.
582 * \param outcnt The output values count.
* \param inary The input values array. The values array and its containing
584 * values are not modified or deallocated by this call.
* \param incnt The input values count.
586 * \return 0 on success or -1 on any error.
*/
588 RMIX_METHOD_CALL(rmiximgproc_imageprocessed_call);

590
/** \fn int rmiximgproc_imageprocessed (char* name);
592 * \brief Deletes a processed image from the internal worklist. If all images
* are processed the plug-in is shutdown.
594 *
* \param *name Name of the image.
596 * \return 0 on success or -1 on any error.
*/
598 int rmiximgproc_imageprocessed (char* name);

600
/** \fn int rmiximgprocclient_imageprocessed_oneway ( \
602 * rmix_remoteref_t *remoteobj, \
* char *name);
604 * \brief Calls a plug-in for informing it that an image was
* successfully stored (client-side method stub).
606 *
* \param *remoteobj ID of the remote object
608 * \param *name Name of the image file
```

A. Appendix

```
* \return          0 on success or -1 on any error
610 */
int rmiximgprocclient_imageprocessed_oneway ( rmix_remoteref_t *remoteobj,
612                                             char                *name);

614
/** \fn          int rmixppmclient_repairpipe ( rmix_remoteref_t *remoteobj,\
616                                             char                *node);
* \brief        Calls the Parallel Plug-in Manager for repairing the pipeline
618 *              (client-side method stub).
*
620 * \param       *remoteobj  ID of the remote object
* \param       *node        Name of the not reachable node
622 * \return      0 on success or -1 on any error
*/
624 int rmixppmclient_repairpipe ( rmix_remoteref_t *remoteobj,
                                char                *node);
626

628 /** \fn          int imgproc_createremoteref( rmix_remoteref_t **remoteobj,\
                                                char                *objectid,\
630                                                char                *node);
* \brief        Creates a remote reference specified by objectid and node. The RPC
632 *              protocol is used.
*
634 * \param       **remoteobj  ID of the remote object
* \param       *objectid     ID of the exported object
636 * \param       *node        Name of the node
* \return      0 on success or -1 on any error
638 */
int imgproc_createremoteref( rmix_remoteref_t **remoteobj,
640                             char            *objectid,
                             char            *node);
642

644 /*****
*
646 * The following section contains the image processing pipeline functions.
*
648 *****/

650 /** \fn          int imgproc_decrease_imagecounter();
* \brief        Decreases the image counter by 1.
652 *
* \return      0 on success or -1 on any error
654 */
int imgproc_decrease_imagecounter();
656

658 /** \fn          int imgproc_create_magickwand( char *imgfile, \
                                                char *imgname);
660 * \brief        Processes a loaded image on the first plug-in unit of the pipeline
*              and forwards it to the next unit.
662 *
* \param       *imgfile     Image name + path
664 * \param       *imgname     Image name
* \return      0 on success or -1 on any error
666 */
int imgproc_create_magickwand( char *imgfile,
668                             char *imgname);

670
/** \fn          int imgproc_filter_rotation( unsigned char **blob, \
```

A. Appendix

```
672                                     size_t      *size);
673 * \brief Rotates an image by 90 degrees.
674 *
675 * \param **blob Char array containing the image data.
676 * \param *size Size of the char array.
677 * \return      0 on success or -1 on any error
678 */
679 int imgproc_filter_rotation( unsigned char **blob,
680                             size_t      *size);
681
682 /** \fn      int imgproc_filter_oilpainting( unsigned char **blob, \
683                                     size_t      *size);
684 * \brief Puts an oil painting filter on an image.
685 *
686 * \param **blob Char array containing the image data.
687 * \param *size Size of the char array.
688 * \return      0 on success or -1 on any error
689 */
690 int imgproc_filter_oilpainting( unsigned char **blob,
691                                 size_t      *size);
692
693 /** \fn      int imgproc_filter_swirl( unsigned char **blob, \
694                                     size_t      *size);
695 * \brief Swirls the center of an image.
696 *
697 * \param **blob Char array containing the image data.
698 * \param *size Size of the char array.
699 * \return      0 on success or -1 on any error
700 */
701 int imgproc_filter_swirl( unsigned char **blob,
702                            size_t      *size);
703
704 /** \fn      int imgproc_filter_negate( unsigned char **blob, \
705                                     size_t      *size);
706 * \brief Negates an image.
707 *
708 * \param **blob Char array containing the image data.
709 * \param *size Size of the char array.
710 * \return      0 on success or -1 on any error
711 */
712 int imgproc_filter_negate( unsigned char **blob,
713                             size_t      *size);
714
715 /** \fn      int imgproc_filter_solarize( unsigned char **blob, \
716                                     size_t      *size);
717 * \brief Puts a solarize filter on an image.
718 *
719 * \param **blob Char array containing the image data.
720 * \param *size Size of the char array.
721 * \return      0 on success or -1 on any error
722 */
723 int imgproc_filter_solarize( unsigned char **blob,
724                               size_t      *size);
725
726 /** \fn      int imgproc_write_image( unsigned char *blob, \
727                                     size_t      size, \
728                                     char      *filename);
729 * \brief Stores an image.
```


A. Appendix

```
*
736 * \param **blob      Char array containing the image data.
* \param  *size      Size of the char array.
738 * \param  *filename  Name of the new image file.
* \return          0 on success or -1 on any error
740 */
int imgproc_write_image( unsigned char *blob ,
742                      size_t      size ,
                      char          *filename);
744

746 /** \fn      int imgproc_get_files( char *dir);
* \brief Returns the number of files found in a directory.
748 *
* \param  *dir      Directory name.
750 * \return          Number of files.
*/
752 int imgproc_get_files( char *dir);

754
756 /** \fn      int imgproc_loadimages( char *imgdir);
* \brief Tries to load the image files in the specified directory and
* additional files.
758 *
* \param  *imgdir   Directory containing the image files.
760 * \return          0 on success or -1 on any error
*/
762 int imgproc_loadimages( char *imgdir);

764
766 /******
* next section contains functions for accessing the image worklist
768 *
*****/
770
772 /** \fn      int imgproc_worklist_insert( char          *name,\
size_t      size,\
unsigned char *blob, \
worklist_t  **list_pointer);
774
* \brief Inserts an element in the worklist.
776 *
* \param  *name      Name of the image
778 * \param  size      Size of the image
* \param  *blob      Data of the image
780 * \param  **list_pointer Pointer to the linked list
* \return          0 on success or -1 on any error
782 */
int imgproc_worklist_insert( char          *name,
784                          size_t      size ,
                          unsigned char *blob ,
                          worklist_t   **list_pointer);
786

788
790 /** \fn      void imgproc_worklist_destroystack( worklist_t **list_pointer);
* \brief Deletes the linked list defined by the pointer.
*
792 * \param  **list_pointer Pointer to the linked list
*/
794 void imgproc_worklist_destroystack( worklist_t **list_pointer);

796
798 /** \fn      int imgproc_worklist_delete( char          *name, \
```

A. Appendix

```
798         worklist_t **list_pointer);
800 * \brief Deletes the element of the list defined by the name.
801 *
802 * \param *name Name of the element.
803 * \param **list_pointer Pointer to the linked list
804 * \return 0 on success otherwise -1
805 */
806 int imgproc_worklist_delete( char *name,
807                             worklist_t **list_pointer);
808
809 /** \fn void imgproc_worklist_printlist( worklist_t *list_pointer);
810 * \brief Prints the names of the list elements.
811 *
812 * \param *list_pointer Pointer to the linked list
813 */
814 void imgproc_worklist_printlist( worklist_t *list_pointer);
815
816
817 /** \fn int imgproc_worklist_find_elementposition(char *entry, \
818                                             worklist_t *list_pointer);
819 * \brief Finds the position of an entry.
820 *
821 * \param *char Entry to find
822 * \param *list_pointer Pointer to the linked list
823 * \return The position or otherwise -1
824 */
825 int imgproc_worklist_find_elementposition( char *entry,
826                                           worklist_t *list_pointer);
827
828
829 /*****
830 *
831 * Data
832 *
833 *****/
834
835 /** \var extern imgproc_t imgproc
836 * \brief Integral plug-in "object".
837 *
838 * An external "object" of the Integral plug-in "class".
839 */
840 extern imgproc_t imgproc;
841
842
843 #endif /* IMGPROC_IMGPROC_H */
844
845 /*****
846 *
847 * END OF FILE
848 *
849 *****/
```

A.3.4.2. Source File

```
2 /*****
3 *
4 * Source file for the image processing module.
5 * Copyright (c) Ronald Baumann.
6 *
7 * For more information see the following files in the source distribution top-
```

A. Appendix

```

 * level directory or package data directory (usually /usr/local/share/package):
8  *
 * - README    for general package information.
10 * - INSTALL  for package install information.
 * - COPYING   for package license information and copying conditions.
12 * - AUTHORS  for package authors information.
 * - ChangeLog for package changes information.
14 *
 *****/
16
/** \file imgproc.c
18 * \brief Sorce file for image processing module.
 *
20 * The libimgproc module is a plug-in for the Harness project. It performs
 * various image filters on pictures.
22 */

24
/*****
26 *
 * Includes
28 *
 *****/
30
/* Main module header file. */
32 #include "imgproc.h"

34
/*****
36 *
 * Data Types
38 *
 *****/
40
/** \struct imgproc_data_t
42 * \brief Image processing plug-in module data type.
 *
44 * Contains the mutex and the instances handler.
 */
46 typedef struct
{
48     pthread_mutex_t  mutex;           /* mutex */
49     unsigned int    count;           /* instances array count */
50     struct
    {
52         unsigned int handle;         /* instance handle */
53     } *instances;                   /* instances array */
54     unsigned int    filter;          /* filter which will be used */
55     char            *sourcedir;      /* source images dir */
56     char            *targetdir;      /* target dir for images */
57     char            *successor;      /* successor plug-in */
58     char            *predecessor;    /* predecessor plug-in */
59     char            *ppmnode;        /* node running the PPM */
60     unsigned int    imagecounter;    /* counter for images to process */
62 }imgproc_data_t;

64
/** \struct imgproc_rmix_data_t
66 * \brief Data for rmix communication functions.
 */
68 typedef struct
{
```

A. Appendix

```
70   rmix_localref_t *localref_harness;    /* exported Harness kernel */
71   rmix_localref_t *localref_imgproc;   /* exported imgproc object */
72   unsigned int    handle_rmix;        /* handle for rmix plug-in */
73 }imgproc_rmix_data_t;
74
75
76 /** \var   int(*pFunc)( unsigned char **, size_t *);
77  * \brief  Definition of a pointer to a image filter function.
78  */
79 typedef int(*pFunc)( unsigned char **, size_t *);
80
81
82 /*****
83  *
84  * Function Prototypes
85  *
86  *****/
87
88 /** \fn    HARNESS_PLUGINS_INIT(imgproc_init);
89  * \brief  Initializes the image processing plug-in.
90  *
91  * \param  handle The plug-in instance handle.
92  * \return 0 on success or -1 on any error with errno set appropriately.
93  */
94 HARNESS_PLUGINS_INIT(imgproc_init);
95
96
97 /** \fn    HARNESS_PLUGINS_FINI(imgproc_fini);
98  * \brief  Finalizes the image processing plug-in.
99  *
100 * \param  handle The plug-in instance handle.
101 * \return 0 on success or -1 on any error with errno set appropriately.
102 */
103 HARNESS_PLUGINS_FINI(imgproc_fini);
104
105
106 /*****
107  *
108  * Data
109  *
110  *****/
111
112 /** \var   rmix_method_t rmixppmclient_methods[RMIKPPM_METHODS_COUNT]
113  * \brief  Client-side method descriptors for rmix ppm.
114  */
115 const rmix_method_t rmixppmclient_methods[RMIKPPM_METHODS_COUNT] =
116     RMIKPPM_METHODS_CLIENT;
117
118
119 /** \var   rmix_interface_t rmixppmclient_interface
120  * \brief  Client-side interface for rmix ppm.
121  */
122 const rmix_interface_t rmixppmclient_interface = {
123     RMIKPPM_METHODS_COUNT, /* method descriptor count */
124     (rmix_method_t*)rmixppmclient_methods /* method descriptor array */
125 };
126
127
128 /** \var   const rmix_method_t rmiximgproc_methods[RMIKIMGPROC_METHODS_COUNT]
129  * \brief  Server-side method descriptors for rmix imgproc.
130  */
131 const rmix_method_t rmiximgproc_methods[RMIKIMGPROC_METHODS_COUNT] =
132     RMIKIMGPROC_METHODS;
```

A. Appendix

```
134 /** \var  const rmix_interface_t rmiximgproc_interface
    * \brief Server-side interface for rmix imgproc.
136 */
    const rmix_interface_t rmiximgproc_interface =
138 {
        RMIXIMGPROC_METHODS_COUNT,          /* method descriptor count */
140        (rmix_method_t*)rmiximgproc_methods /* method descriptor array */
    };
142

144 /** \var  rmix_method_t rmiximgprocclient_methods [RMIXIMGPROC_METHODS_COUNT]
    * \brief Client-side method descriptors for rmix imgproc.
146 */
    const rmix_method_t rmiximgprocclient_methods [RMIXIMGPROC_METHODS_COUNT] =
148        RMIXIMGPROC_METHODS_CLIENT;

150
152 /** \var  const rmix_interface_t rmiximgprocclient_interface
    * \brief Client-side interface for rmix imgproc.
    */
154 const rmix_interface_t rmiximgprocclient_interface = {
        RMIXIMGPROC_METHODS_COUNT,          /* method descriptor count */
156        (rmix_method_t*)rmiximgprocclient_methods /* method descriptor array */
    };
158

160 /** \var  pFunc filters
    * \brief Array with pointers to the image filter functions.
162 */
    const pFunc filters [FILTER_NUMBER] =
164 {
        &imgproc_filter_rotation ,
166        &imgproc_filter_oilpainting ,
        &imgproc_filter_swirl ,
168        &imgproc_filter_negate ,
        &imgproc_filter_solarize
170 };

172
174 /** \var  imgproc_rmix_data_t imgproc_rmix_data
    * \brief Data for rmix communication functions.
    *
176 * Includes the local references to the exported objects.
    */
178 imgproc_rmix_data_t imgproc_rmix_data =
    {
180        NULL,          /* exported Harness kernel */
        NULL,          /* exported imgproc object */
182        0,            /* handle for rmix plug-in */
    };
184

186 /** \var  imgproc_data_t imgproc_data
    * \brief image processing plug-in module data.
188 *
    * Includes the mutex for instances handle and the instances array where the
190 * handles are stored.
    */
192 imgproc_data_t imgproc_data =
    {
194        PTHREAD_MUTEX_INITIALIZER, /* mutex */
        0,                          /* instances array count */
    }
```

A. Appendix

```
196     NULL,                /* instances array      */
197     0,                   /* filter which will be used */
198     NULL,                /* source images dir    */
199     NULL,                /* target dir for images */
200     NULL,                /* successor plug-in    */
201     NULL,                /* predecessor plug-in   */
202     0,                   /* images to process     */
203 };
204
205 /** \var  worklist_t worklist
206 *  \brief list with image information of the processed images.
207 */
208 */
209 worklist_t *worklist = NULL;
210
211 unsigned int testswitch = 0;
212
213
214 /** \var  imgproc_t imgproc
215 *  \brief Image processing plug-in "object".
216 *
217 *  Contains the version of the library and possible public data and function
218 *  pointers.
219 */
220 */
221 imgproc_t imgproc =
222 {
223     {
224         LIBIMGPROC_VERSION_CURRENT, /* current version      */
225         LIBIMGPROC_VERSION_REVISION, /* current revision     */
226         LIBIMGPROC_VERSION_AGE,     /* version age          */
227         LIBIMGPROC_VERSION_FIRST    /* first supported version */
228     }
229     /* plug-in version */
230 };
231
232 /*****
233 *
234 * Functions
235 *
236 *****/
237
238 /*
239 * Initializes the image processing plug-in plug-in.
240 *
241 * handle = The plug-in instance handle.
242 * return = 0 on success or -1 on any error with errno set appropriately.
243 */
244 #undef __FUNC__
245 #define __FUNC__ "imgproc_init"
246 HARNESS_PLUGINS_INIT(imgproc_init)
247 {
248     int          error;
249     void         *instances;
250     unsigned int index;
251
252     IMGPROC_PRINT((
253         IMGPROC_INFO(libimgproc is starting)))
254
255     /* Lock image processing plug-in mutex. */
256     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex)))
257     {
258         errno = error;
```

A. Appendix

```
260     IMGPROC_PRINT((
        IMGPROC_WARN(unable to lock image processing plug-in mutex)))
        harness_syserr();
262     return -1;
    }
264
    /* Search for handle in instances array. */
266     for (index = 0; index < imgproc_data.count; index++)
    {
268         if (imgproc_data.instances[index].handle == handle)
        {
270             IMGPROC_PRINT((
                IMGPROC_WARN(handle is already in instances array)))
                harness_syserr();
272
                /* Unlock image processing plug-in mutex. */
274             if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
                {
276                 errno = error;
                IMGPROC_PRINT((
278                     IMGPROC_WARN(unable to unlock image processing plug-in
                                mutex)))
                                harness_syserr();
282             }
284             return -1;
        }
286     }
288
    /* Increase instances array. */
    index = imgproc_data.count;
290     imgproc_data.count++;
    /* Reallocate instances array. */
292     if (NULL == (instances = realloc( imgproc_data.instances ,
                                        imgproc_data.count *
294                                         sizeof( imgproc_data.instances [0]))))
    {
296         IMGPROC_PRINT((
            IMGPROC_WARN(unable to reallocate instances array)))
            /* Unlock image processing plug-in mutex. */
            if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
            {
300                 errno = error;
                IMGPROC_PRINT((
302                     IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
                    harness_syserr();
304             }
            return -1;
306     }
308
    imgproc_data.instances = instances;
310     /* Save instances array entry. */
    imgproc_data.instances[index].handle = handle;
312     /* Check for first initialization. */
    if (1 == imgproc_data.count)
314     {
        /******
316         /* PUT YOUR INIT CODE HERE */
        /******
318         if (imgproc_init_rmix() != 0)
        {
320             /* if the communication interface cannot be exported terminate
                the plug-in */

```

A. Appendix

```
322     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
323     {
324         errno = error;
325         IMGPROC_PRINT((
326             IMGPROC_WARN(unable to unlock image processing plug-in
327                           mutex)))
328         harness_syserr();
329         return -1;
330     }
331     harness_kernel_shutdown();
332 }
333
334 /* Unlock image processing plug-in mutex. */
335 if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
336 {
337     errno = error;
338     IMGPROC_PRINT((
339         IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
340     harness_syserr();
341     return -1;
342 }
343 return 0;
344 }
345
346
347 /*
348  * Finalizes the image processing plug-in.
349  *
350  * handle = The plug-in instance handle.
351  * return = 0 on success or -1 on any error with errno set appropriately.
352  */
353 #undef __FUNC__
354 #define __FUNC__ "imgproc_fini"
355 HARNESS_PLUGINS_FINI(imgproc_fini)
356 {
357     int error;
358     void *instances;
359     unsigned int index;
360
361     IMGPROC_PRINT((
362         IMGPROC_INFO(libimgproc is shutting down)))
363
364     /* Lock plug-in mutex. */
365     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex)))
366     {
367         errno = error;
368         IMGPROC_PRINT((
369             IMGPROC_WARN(unable to lock image processing plug-in mutex)))
370         harness_syserr();
371         return -1;
372     }
373
374     /* Search for handle in instances array. */
375     for (index = 0; index < imgproc_data.count; index++)
376     {
377         if (imgproc_data.instances[index].handle == handle)
378         {
379             break;
380         }
381     }
382     /* Check if handle is not in instances array. */
383     if (index == imgproc_data.count)
```


A. Appendix

```
386 {
    IMGPROC_PRINT((
388     IMGPROC_WARN(handle is not in instances array)))
    harness_syserr();
    /* Unlock harness-example plug-in mutex. */
390     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
    {
392         errno = error;
        IMGPROC_PRINT((
394             IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
        harness_syserr();
396     }
    return -1;
398 }

400 /* Remove instance from instances array. */
imgproc_data.count--;
402 memmove(imgproc_data.instances + index,
        imgproc_data.instances + index + 1,
404         (imgproc_data.count - index) *
            sizeof(unsigned int));
406
    /* Reallocate instances array. */
408     if (0 == imgproc_data.count)
    {
410         free(imgproc_data.instances);
        imgproc_data.instances = NULL;
412     }
    else if (NULL == (instances = realloc( imgproc_data.instances,
414                                         imgproc_data.count *
                                            sizeof( imgproc_data.instances[0]))))
416     {
418         IMGPROC_PRINT((
            IMGPROC_WARN(unable to reallocate instances array)))
        }
420     else
    {
422         imgproc_data.instances = instances;
    }
424 /* Check for last finalization. */
    if (0 == imgproc_data.count)
426     {
        /******
428         /* PUT YOUR FINI CODE HERE */
        /******
430         imgproc_worklist_destroylist( &worklist);
        imgproc_fini_rmix();
432     }

434 /* Unlock image processing plug-in mutex. */
    if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
436     {
438         errno = error;
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
        harness_syserr();
440         return -1;
442     }
    return 0;
444 }

446 /*
```

A. Appendix

```
448 *   Initializes the necessary rmix parameter and exports the plug-ins.
449 *
450 *   \return 0 on success or -1 on any error
451 */
452 #undef __FUNC__
453 #define __FUNC__ "imgproc_init_rmix"
454 int imgproc_init_rmix()
455 {
456     int ret;
457
458     IMGPROC_PRINT((
459         IMGPROC_INFO(start initialization of RMIX)))
460
461     /* Load RMIX plug-in and initialize RMIX */
462     ret = harness_plugins_load( &imgproc_rmix_data.handle_rmix,
463                               "libharness-rmix.0", HARNESS_PLUGINS_EXPORT);
464
465     if (ret != 0)
466     {
467         IMGPROC_PRINT((
468             IMGPROC_WARN(unable to load RMIX plug-in)))
469         return -1;
470     }
471
472     /* Exports Harness while forcing 1000 as specific object handle */
473     ret = harness_rmix_export4( &imgproc_rmix_data.localref_harness,
474                               "PROTOCOL=RPC OBJECTID=1000");
475
476     if (ret != 0)
477     {
478         RMIX_LOG((RMIX_WARN(unable to export Harness kernel)))
479         return -1;
480     }
481
482     /* Exports imgproc plug-in while forcing 1002 as specific object handle */
483     ret = rmix_export4( &imgproc_rmix_data.localref_imgproc,
484                       "PROTOCOL=RPC OBJECTID=1002", NULL, &miximgproc_interface);
485
486     if (ret != 0)
487     {
488         RMIX_LOG((RMIX_WARN(unable to export imgproc interface)))
489         return -1;
490     }
491
492     IMGPROC_PRINT((
493         IMGPROC_INFO(RMIX initialized and imgproc plug-in exported)))
494
495     return 0;
496 }
497
498 /*
499 *   Finalizes rmix and unexports the plug-ins.
500 *
501 *   \return 0 on success or -1 on any error
502 */
503 #undef __FUNC__
504 #define __FUNC__ "imgproc_fini_rmix"
505 int imgproc_fini_rmix()
506 {
507     int ret;
508     int err = 0;
509
510     IMGPROC_PRINT((
```

A. Appendix

```

    IMGPROC_INFO(start finalization of RMIX))
512
    /* Unexports imgproc plug-in */
514    ret = rmix_unexport( &imgproc_rmix_data.localref_imgproc);
    if (ret != 0)
516    {
        RMIX_LOG((RMIX_WARN(unable to unexport imgproc interface)))
518        err = -1;
    }
520
    /* Unexports Harness */
522    ret = harness_rmix_unexport( &imgproc_rmix_data.localref_harness);
    if (ret != 0)
524    {
        RMIX_LOG((RMIX_WARN(unable to unexport Harness kernel)))
526        err = -1;
    }
528
    /* Unloads RMIX plug-in */
530    ret = harness_plugins_unload( imgproc_rmix_data.handle_rmix);
    if (ret != 0)
532    {
        IMGPROC_PRINT((
534            IMGPROC_WARN(unable to unload RMIX plug-in)))
        err = -1;
536    }

538    IMGPROC_PRINT((
540        IMGPROC_INFO(RMIX finalized and imgproc plug-in unexported)))
    return err;
542 }

544
546 /*
548 * next section contains functions for communication
550 *
552 /*
554 * Accepts calls for the initialisation of necessary pipeline parameters.
556 * (server-side stub)
558 *
560 * \param object The local object.
562 * \param outary The output values array. The values array and its
564 * containing values are allocated before the call with the
566 * exception of variable arrays and strings. They are allocated
568 * dynamically or explicitly set to NULL by this function. In
570 * the case of variable arrays, the length value is allocated
572 * before the call and is set to 0 if its variable array is
574 * NULL.
576 * \param outcnt The output values count.
578 * \param inary The input values array. The values array and its containing
580 * values are not modified or deallocated by this call.
582 * \param incnt The input values count.
584 * \return 0 on success or -1 on any error.
586 */
#define __FUNC__
570 #define __FUNC__ "rmiximgproc_initpipeline_call"
RMIX_METHOD_CALL(rmiximgproc_initpipeline_call)
572 {
    int result = 0;
```

A. Appendix

```
574 unsigned int filter; /* filter which will be used */
576 char *sourcedir; /* source images dir */
578 char *targetdir; /* target dir for images */
578 char *successor; /* successor plug-in */
578 char *predecessor; /* predecessor plug-in */
580 char *ppmnode; /* node running the PPM */

582 #ifdef DEBUG
582 /* Check object parameter. */
584 /*
586 if (NULL == object)
586 {
588     errno = EINVAL;
588     RMIX_LOG((RMIX_WARN(object parameter is null)))
588     errno = EINVAL;
590     return -1;
592 }
592 */
594 /* Check outary parameter. */
594 if ((NULL == outary)&&(0 != outcnt))
596 {
596     errno = EINVAL;
596     RMIX_LOG((RMIX_WARN(outary parameter is null)))
598     errno = EINVAL;
598     return -1;
600 }

602 /* Check outcnt parameter. */
602 if ((0 == outcnt)&&(NULL != outary))
604 {
606     errno = EINVAL;
606     RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
608     errno = EINVAL;
608     return -1;
610 }

612 /* Check inary parameter. */
612 if ((NULL == inary)&&(0 != incnt))
614 {
614     errno = EINVAL;
614     RMIX_LOG((RMIX_WARN(inary parameter is null)))
616     errno = EINVAL;
616     return -1;
618 }

620 /* Check incnt parameter. */
620 if ((0 == incnt)&&(NULL != inary))
622 {
624     errno = EINVAL;
624     RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
626     errno = EINVAL;
626     return -1;
628 }
628 #endif /* DEBUG */

630 /* Prepare input. */
630 filter = *(unsigned int*)inary[0];
632 sourcedir = (char*) inary[1];
632 targetdir = (char*) inary[2];
634 successor = (char*) inary[3];
634 predecessor = (char*) inary[4];
636 ppmnode = (char*) inary[5];
```

A. Appendix

```
638 /* Call method function. */
639 result = rmixmapproc_initpipeline( filter , sourcedir , targetdir , successor ,
640                                   predecessor , ppmnode);
641
642 /* Prepare output. */
643 *(int*)(outary[0]) = result;
644 /* Reset errno if needed. */
645 if (0 != result)
646 {
647     errno = 0;
648 }
649 return 0;
650 }
651
652 /*
653 * Initializes the necessary pipeline parameters.
654 *
655 * \param filter Filter which will be used
656 * \param *sourcedir Source images dir
657 * \param *targetdir Target dir for images
658 * \param *sucessor Successor plug-in
659 * \param *predecessor Predecessor plug-in
660 * \param *ppmnode Node running the PPM
661 * \return 0 on success or -1 on any error.
662 */
663 #undef __FUNC__
664 #define __FUNC__ "rmixmapproc_initpipeline"
665 int rmixmapproc_initpipeline (unsigned int filter ,
666                               char *sourcedir ,
667                               char *targetdir ,
668                               char *sucessor ,
669                               char *predecessor ,
670                               char *ppmnode)
671 {
672
673     /* copy the values */
674     /* which filter will be used */
675     imgproc_data.filter = filter;
676
677     /* the source directory of the images */
678     imgproc_data.sourcedir = (char*) malloc( sizeof(char) * (strlen(sourcedir)
679                                                         + 1) );
680     if (imgproc_data.sourcedir == NULL)
681     {
682         IMGPROC_PRINT((
683             IMGPROC_WARN(could not allocate memory)))
684         return -1;
685     }
686     strcpy( imgproc_data.sourcedir , sourcedir);
687
688     /* the target directory of the images */
689     imgproc_data.targetdir = (char*) malloc( sizeof(char) * (strlen(targetdir)
690                                                         + 1) );
691     if (imgproc_data.targetdir == NULL)
692     {
693         IMGPROC_PRINT((
694             IMGPROC_WARN(could not allocate memory)))
695         return -1;
696     }
697     strcpy( imgproc_data.targetdir , targetdir);
```

A. Appendix

```
700  /* the successor plug-in unit */
imgproc_data.successor = (char*) malloc( sizeof(char) * (strlen(successor)
702                                     + 1) );
703  if (imgproc_data.successor == NULL)
704  {
705      IMGPROC_PRINT((
706          IMGPROC_WARN(could not allocate memory)))
707      return -1;
708  }
strcpy( imgproc_data.successor, successor);
710
711  /* the predecessor plug-in unit */
imgproc_data.predecessor = (char*) malloc( sizeof(char) *
712                                           (strlen(predecessor) + 1) );
713  if (imgproc_data.predecessor == NULL)
714  {
715      IMGPROC_PRINT((
716          IMGPROC_WARN(could not allocate memory)))
717      return -1;
718  }
strcpy( imgproc_data.predecessor, predecessor);
720
721  /* the node, which runs the PPM */
imgproc_data.ppmnode = (char*) malloc( sizeof(char) *
722                                       (strlen(ppmnode) + 1) );
723  if (imgproc_data.ppmnode == NULL)
724  {
725      IMGPROC_PRINT((
726          IMGPROC_WARN(could not allocate memory)))
727      return -1;
728  }
strcpy( imgproc_data.ppmnode, ppmnode);
730
731  return 0;
732 }
733 }
734 }
735
736
737 /*
738 * Accepts calls for the initialisation of the image counter. (server-side
739 * stub)
740 *
741 * \param object The local object.
742 * \param outary The output values array. The values array and its
743 *               containing values are allocated before the call with the
744 *               exception of variable arrays and strings. They are allocated
745 *               dynamically or explicitly set to NULL by this function. In
746 *               the case of variable arrays, the length value is allocated
747 *               before the call and is set to 0 if its variable array is
748 *               NULL.
749 * \param outcnt The output values count.
750 * \param inary The input values array. The values array and its containing
751 *              values are not modified or deallocated by this call.
752 * \param incnt The input values count.
753 * \return      0 on success or -1 on any error.
754 */
755 #undef __FUNC__
756 #define __FUNC__ "rmiximgproc_setimagecounter_call"
RMIX_METHOD_CALL(rmiximgproc_setimagecounter_call)
757 {
758     int result = 0;
759     unsigned int counter = 0;
760
761 #ifdef DEBUG
762
```

A. Appendix

```
764 /*
    /* Check object parameter. */
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
772 */
    /* Check outary parameter. */
774 if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }
780
782 /* Check outcnt parameter. */
784 if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }
790
792 /* Check inary parameter. */
794 if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }
798
800 /* Check incnt parameter. */
802 if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
806
808 #endif /* DEBUG */

810 /* Prepare input. */
    counter = *(unsigned int*)inary[0];
812
    /* Call method function. */
814 result = rmixingproc_setimagecounter( counter);

816 /* Prepare return 0; output. */
    *(int*)(outary[0]) = result;
818
    /* Reset errno if needed. */
820 if (0 != result)
    {
        errno = 0;
    }
824 return 0;
}
```

A. Appendix

```
826
828 /*
829  * Forwards an image to the next plug-in.
830  *
831  * \param counter Number of images to process
832  * \return         0 on success or -1 on any error.
833  */
834 #undef __FUNC__
835 #define __FUNC__ "rmiximgproc_setimagecounter"
836 int rmiximgproc_setimagecounter( unsigned int counter)
837 {
838     rmix_remoteref_t *remoteobj;
839     int ret;
840
841     imgproc_data.imagecounter = counter;
842
843     /* if the plug-in has a successor, the image counter has to be forwarded
844      * to it */
845     if (strcmp(imgproc_data.successor, "nada") != 0)
846     {
847         /* prepare the remote reference of the successor */
848         ret = imgproc_createremoteref( &remoteobj, "1002",
849                                         imgproc_data.successor);
850         if (ret != 0)
851         {
852             IMGPROC_PRINT((
853                 IMGPROC_WARN(could not create remote object references)))
854             return -1;
855         }
856
857         /* forward the image counter to the successor */
858         ret = rmiximgprocclient_setimagecounter ( remoteobj, counter);
859         if (ret != 0)
860         {
861             IMGPROC_PRINT((
862                 IMGPROC_WARN(could not set image counter of node %s),
863                 imgproc_data.successor))
864             return -1;
865         }
866
867         /* delete the remote reference */
868         ret = rmix_remoteref_destroy( &remoteobj);
869         if (ret != 0)
870         {
871             IMGPROC_PRINT((
872                 IMGPROC_WARN(could not destroy remote object references)))
873             }
874         }
875     }
876     return 0;
877 }
878
879
880 /*
881  * Accepts calls for updating the image counter. (server-side stub)
882  *
883  * \param object The local object.
884  * \param outary The output values array. The values array and its
885  *               containing values are allocated before the call with the
886  *               exception of variable arrays and strings. They are allocated
887  *               dynamically or explicitly set to NULL by this function. In
888  *               the case of variable arrays, the length value is allocated
```


A. Appendix

```

*           before the call and is set to 0 if its variable array is
890 *           NULL.
*   \param outcnt The output values count.
892 *   \param inary  The input values array. The values array and its containing
*           values are not modified or deallocated by this call.
894 *   \param incnt  The input values count.
*   \return        0 on success or -1 on any error.
896 */
#undef __FUNC__
898 #define __FUNC__ "rmiximgproc_updateimagecounter_call"
RMIX_METHOD_CALL(rmiximgproc_updateimagecounter_call)
900 {
    int          result = 0;
902    unsigned int counter = 0;

904 #ifdef DEBUG
    /* Check object parameter. */
906 /*
    if (NULL == object)
908     {
        errno = EINVAL;
910         RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
912         return -1;
    }
914 */
    /* Check outary parameter. */
916 if ((NULL == outary)&&(0 != outcnt))
    {
918         errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
920         errno = EINVAL;
        return -1;
922     }

924 /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
926     {
928         errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
930         errno = EINVAL;
        return -1;
    }
932
    /* Check inary parameter. */
934 if ((NULL == inary)&&(0 != incnt))
    {
936         errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
938         errno = EINVAL;
        return -1;
940     }

942 /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
944     {
946         errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
948         errno = EINVAL;
        return -1;
    }
950 #endif /* DEBUG */

```

A. Appendix

```
952  /* Prepare input. */
     counter = *(unsigned int*)inary[0];
954
     /* Call method function. */
956  result = rmiximgproc_updateimagecounter( counter);
958
     /* Prepare return 0; output. */
     *(int*)(outary[0]) = result;
960
     /* Reset errno if needed. */
962  if (0 != result)
     {
964      errno = 0;
     }
966
     return 0;
968 }

970
     /*
972  * Updates the image counter after the pipeline was broken.
     *
974  * \param counter Number of images to process
     * \return 0 on success or -1 on any error.
976  */
     #undef __FUNC__
978 #define __FUNC__ "rmiximgproc_updateimagecounter"
     int rmiximgproc_updateimagecounter( unsigned int counter)
980 {
     imgproc_data.imagecounter = counter;
982
     return 0;
984 }

986
     /*
988  * Calls a defined plug-in unit for setting its image counter (client-side
     * method stub).
990  *
     * \param *remoteobj ID of the remote object
992  * \param counter Image counter
     * \return 0 on success or -1 on any error
994  */
     #undef __FUNC__
996 #define __FUNC__ "rmiximgprocclient_setimagecounter"
     int rmiximgprocclient_setimagecounter ( rmix_remoteref_t *remoteobj,
998                                         unsigned int counter)
     {
1000  const void *inary[1];
     void *outary[1];
1002  int result;

1004  /* Prepare parameters. */
     inary[0] = &counter;
1006  outary[0] = &result;

1008  /* Invoke remote object method. */
     if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
1010                          RMIXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX, outary, 1,
                          inary, 1))
1012  {
     int errno2 = errno;
1014  RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))

```

A. Appendix

```
        errno = errno2;
1016     return -1;
    }
1018
    return result;
1020 }

1022
/*
1024 * Accepts calls for the invocation of the pipeline. (server-side stub)
    *
1026 * \param object The local object.
    * \param outary The output values array. The values array and its
1028 *               containing values are allocated before the call with the
    *               exception of variable arrays and strings. They are allocated
1030 *               dynamically or explicitly set to NULL by this function. In
    *               the case of variable arrays, the length value is allocated
1032 *               before the call and is set to 0 if its variable array is
    *               NULL.
1034 * \param outcnt The output values count.
    * \param inary The input values array. The values array and its containing
1036 *               values are not modified or deallocated by this call.
    * \param incnt The input values count.
1038 * \return      0 on success or -1 on any error.
    */
1040 #undef __FUNC__
    #define __FUNC__ "rmiximgproc_invokepipeline_call"
1042 RMX_METHOD_CALL(rmiximgproc_invokepipeline_call)
    {
1044     int         result;

1046 #ifdef DEBUG
        /* Check object parameter. */
1048 /*
        if (NULL == object)
1050     {
            errno = EINVAL;
1052             RMX_LOG((RMX_WARN(object parameter is null)))
            errno = EINVAL;
1054             return -1;
        }
1056 */

1058 /* Check outary parameter. */
        if ((NULL == outary)&&(0 != outcnt))
1060     {
            errno = EINVAL;
1062             RMX_LOG((RMX_WARN(outary parameter is null)))
            errno = EINVAL;
1064             return -1;
        }
1066
        /* Check outcnt parameter. */
1068 if ((0 == outcnt)&&(NULL != outary))
        {
1070             errno = EINVAL;
            RMX_LOG((RMX_WARN(outcnt parameter is zero)))
1072             errno = EINVAL;
            return -1;
1074         }

1076 /* Check inary parameter. */
        if ((NULL == inary)&&(0 != incnt))
```

A. Appendix

```
1078     {
1080         errno = EINVAL;
1080         RMIX_LOG((RMIX_WARN(inary parameter is null)))
1080         errno = EINVAL;
1082         return -1;
1082     }
1084
1084     /* Check incnt parameter. */
1086     if ((0 == incnt)&&(NULL != inary))
1086     {
1088         errno = EINVAL;
1088         RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
1088         errno = EINVAL;
1088         return -1;
1088     }
1092 #endif /* DEBUG */
1094
1094     /* Call method function. */
1096     result = rmiximgproc_invokepipeline();
1096
1098     /* Prepare output. */
1098     *(int*)(outary[0]) = result;
1098
1100     /* Reset errno if needed. */
1102     if (0 != result)
1102     {
1104         errno = 0;
1104     }
1106     return 0;
1106 }
1108
1110 /*
1110  * Starts the image processing pipeline.
1112  *
1112  * \return 0 on success or -1 on any error.
1114  */
1114 #undef __FUNC__
1116 #define __FUNC__ "rmiximgproc_invokepipeline"
1116 int rmiximgproc_invokepipeline ()
1118 {
1118     int ret;
1118
1120     /* load the images, use the first filter and forwards the image to a
1122     possible successor plug-in */
1122     ret = imgproc_loadimages(imgproc_data.sourcedir);
1124     if (ret != 0)
1124     {
1126         IMGPROC_PRINT((
1126             IMGPROC_WARN(execution of image processing pipeline not possible)))
1128         return -1;
1128     }
1130
1130     return 0;
1132 }
1134
1134 /*
1136  * Accepts calls, which check if the plug-in was loaded (server-side
1136  * method stub).
1138  *
1138  * \param object The local object.
1140  * \param outary The output values array. The values array and its
```

A. Appendix

```

1142 *           containing values are allocated before the call with the
1143 *           exception of variable arrays and strings. They are allocated
1144 *           dynamically or explicitly set to NULL by this function. In
1145 *           the case of variable arrays, the length value is allocated
1146 *           before the call and is set to 0 if its variable array is
1147 *           NULL.
1148 *           \param outcnt The output values count.
1149 *           \param inary  The input values array. The values array and its containing
1150 *           values are not modified or deallocated by this call.
1151 *           \param incnt  The input values count.
1152 *           \return       0 on success or -1 on any error.
1153 */
1154 RMIX_METHOD_CALL(rmixingproc_availabilitycheck_call)
1155 {
1156     int result = 0;
1157 #ifdef DEBUG
1158     /* Check object parameter. */
1159     /*
1160     if (NULL == object)
1161     {
1162         errno = EINVAL;
1163         RMIX_LOG((RMIX_WARN(object parameter is null)))
1164         errno = EINVAL;
1165         return -1;
1166     }
1167     */
1168     /* Check outary parameter. */
1169     if ((NULL == outary)&&(0 != outcnt))
1170     {
1171         errno = EINVAL;
1172         RMIX_LOG((RMIX_WARN(outary parameter is null)))
1173         errno = EINVAL;
1174         return -1;
1175     }
1176     /* Check outcnt parameter. */
1177     if ((0 == outcnt)&&(NULL != outary))
1178     {
1179         errno = EINVAL;
1180         RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
1181         errno = EINVAL;
1182         return -1;
1183     }
1184     /* Check inary parameter. */
1185     if ((NULL == inary)&&(0 != incnt))
1186     {
1187         errno = EINVAL;
1188         RMIX_LOG((RMIX_WARN(inary parameter is null)))
1189         errno = EINVAL;
1190         return -1;
1191     }
1192     /* Check incnt parameter. */
1193     if ((0 == incnt)&&(NULL != inary))
1194     {
1195         errno = EINVAL;
1196         RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
1197         errno = EINVAL;
1198         return -1;
1199     }
1200 }
1201
```

A. Appendix

```
1204 #endif /* DEBUG */

1206     /* Prepare output. */
1207     *(int*)(outary[0]) = result;
1208
1209     return 0;
1210 }

1212
1213 /*
1214 * Accepts call , which forward images to the plug-in. (server-side stub)
1215 *
1216 * \param object The local object.
1217 * \param outary The output values array. The values array and its
1218 *               containing values are allocated before the call with the
1219 *               exception of variable arrays and strings. They are allocated
1220 *               dynamically or explicitly set to NULL by this function. In
1221 *               the case of variable arrays , the length value is allocated
1222 *               before the call and is set to 0 if its variable array is
1223 *               NULL.
1224 * \param outcnt The output values count.
1225 * \param inary The input values array. The values array and its containing
1226 *              values are not modified or deallocated by this call.
1227 * \param incnt The input values count.
1228 * \return      0 on success or -1 on any error.
1229 */
1230 #undef __FUNC__
1231 #define __FUNC__ "rmiximgproc_passimage_call"
1232 RMIX_METHOD_CALL(rmiximgproc_passimage_call)
1233 {
1234     int         result;
1235     char        *name;
1236     unsigned char *blob;
1237     unsigned int bytes;
1238
1239     #ifdef DEBUG
1240     /* Check object parameter. */
1241     /*
1242     if (NULL == object)
1243     {
1244         errno = EINVAL;
1245         RMIX_LOG((RMIX_WARN(object parameter is null)))
1246         errno = EINVAL;
1247         return -1;
1248     }
1249     */
1250
1251     /* Check outary parameter. */
1252     if ((NULL == outary)&&(0 != outcnt))
1253     {
1254         errno = EINVAL;
1255         RMIX_LOG((RMIX_WARN(outary parameter is null)))
1256         errno = EINVAL;
1257         return -1;
1258     }
1259
1260     /* Check outcnt parameter. */
1261     if ((0 == outcnt)&&(NULL != outary))
1262     {
1263         errno = EINVAL;
1264         RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
1265         errno = EINVAL;
1266         return -1;
```

A. Appendix

```

    }
1268
    /* Check inary parameter. */
1270 if ((NULL == inary)&&(0 != incnt))
    {
1272     errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
1274     errno = EINVAL;
        return -1;
1276 }

1278 /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
1280     {
1282         errno = EINVAL;
            RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
1284         errno = EINVAL;
            return -1;
    }
1286 #endif /* DEBUG */

1288     name = (char*)      inary[0];
        bytes = *(unsigned int*) inary[1];
1290     blob = (unsigned char*)inary[2];

1292     /* Call method function. */
        result = rmiximgproc_passimage( name, bytes, blob);
1294
        /* Prepare output. */
1296     *(int*)(outary[0]) = result;

1298     /* Reset errno if needed. */
        if (0 != result)
1300     {
1302         errno = 0;
    }
        return 0;
1304 }

1306
    /*
1308 * Processes an image and forwards it to the next plug-in unit if possible.
    *
1310 * \param *name Name of the image file
    * \param size Size of image data
1312 * \param *blob Image data
    * \return 0 on success or -1 on any error.
1314 */
    #undef __FUNC__
1316 #define __FUNC__ "rmiximgproc_passimage"
    int rmiximgproc_passimage (char      *name,
1318                             size_t    size,
                                unsigned char *blob)
1320 {
        char      *storename;
1322     char      *tmpfile;
        rmix_remoteref_t *remoteobj;
1324     rmix_remoteref_t *remoteobj_ppm;

1326     int error;
        int ret;
1328
        /* process filter */

```

A. Appendix

```
1330 ret = filters[imgproc_data.filter](&blob, &size);
1331 if (ret != 0)
1332 {
1333     IMGPROC_PRINT((
1334         IMGPROC_WARN(could not process image filter)))
1335     return -1;
1336 }
1337
1338 /* if a successor plug-in exists, forward the image to it */
1339 if (strcmp(imgproc_data.successor, "nada") != 0)
1340 {
1341     /* prepare the remote reference of the successor */
1342     ret = imgproc_createremoteref( &remoteobj, "1002",
1343                                     imgproc_data.successor);
1344     if (ret != 0)
1345     {
1346         IMGPROC_PRINT((
1347             IMGPROC_WARN(could not create remote object references)))
1348
1349         /* prepare the remote reference of the PPM node */
1350         ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
1351                                         imgproc_data.ppmnode);
1352         if (ret != 0)
1353         {
1354             IMGPROC_PRINT((
1355                 IMGPROC_WARN(could not create remote object references)))
1356             return -1;
1357         }
1358
1359         /* call the PPM for repairing the pipeline */
1360         ret = rmixppmclient_repairpipe( remoteobj_ppm,
1361                                         imgproc_data.successor);
1362         if (ret != 0)
1363         {
1364             IMGPROC_PRINT((
1365                 IMGPROC_WARN(could not repair pipe)))
1366             rmix_remoteref_destroy( &remoteobj_ppm);
1367             return -1;
1368         }
1369
1370         rmix_remoteref_destroy( &remoteobj_ppm);
1371         ret = imgproc_createremoteref( &remoteobj, "1002",
1372                                         imgproc_data.successor);
1373         if (ret != 0)
1374         {
1375             IMGPROC_PRINT((
1376                 IMGPROC_WARN(could not create remote object
1377                             references)))
1378             return -1;
1379         }
1380     }
1381
1382     /* pass the image with an onway RPC call */
1383     ret = rmiximgprocclient_passimage_oneway ( remoteobj, name, blob, size);
1384     if (ret != 0)
1385     {
1386         IMGPROC_PRINT((
1387             IMGPROC_WARN(could not call remote object for passing image)))
1388
1389         /* prepare the remote reference of the PPM node */
1390         ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
1391                                         imgproc_data.ppmnode);
1392     }
```


A. Appendix

```
1394     if (ret != 0)
1395     {
1396         IMGPROC_PRINT((
1397             IMGPROC_WARN(could not create remote object references)))
1398         return -1;
1399     }
1400     /* call the PPM for repairing the pipeline */
1401     ret = rmixppmclient_repairpipe( remoteobj_ppm,
1402                                     imgproc_data.successor);
1403     if (ret != 0)
1404     {
1405         IMGPROC_PRINT((
1406             IMGPROC_WARN(could not repair pipe)))
1407         rmix_remoteref_destroy( &remoteobj_ppm);
1408         return -1;
1409     }
1410     rmix_remoteref_destroy( &remoteobj_ppm);
1411     ret = imgproc_createremoteref( &remoteobj, "1002",
1412                                     imgproc_data.successor);
1413     if (ret != 0)
1414     {
1415         IMGPROC_PRINT((
1416             IMGPROC_WARN(could not create remote object
1417                             references)))
1418         return -1;
1419     }
1420     ret = rmiximgprocclient_passimage_oneway ( remoteobj, name, blob,
1421                                                 size);
1422     if (ret != 0)
1423     {
1424         IMGPROC_PRINT((
1425             IMGPROC_WARN(could not call remote object for passing
1426                             image)))
1427         return -1;
1428     }
1429 }
1430
1431 /* destroy the remote reference */
1432 ret = rmix_remoteref_destroy( &remoteobj);
1433 if (ret != 0)
1434 {
1435     IMGPROC_PRINT((
1436         IMGPROC_WARN(could not destroy remote object references)))
1437 }
1438
1439
1440 /* if image was sent successfully, store it in the internal list until
1441 an acknowledgment arrives */
1442
1443 /* the access to the list is controlled by a mutex */
1444 /* Lock image processing plug-in mutex. */
1445 if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
1446     errno = error;
1447     IMGPROC_PRINT((
1448         IMGPROC_WARN(unable to lock image processing plug-in mutex)))
1449     harness_syserr();
1450     return -1;
1451 }
1452
1453 /* store the image if it was not stored yet */
1454 ret = imgproc_worklist_find_elementposition( name, worklist);
```

A. Appendix

```
1456     if (ret == -1)
1457         imgproc_worklist_insert( name, size, blob, &worklist);
1458
1459     /* Unlock image processing plug-in mutex. */
1460     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
1461     {
1462         errno = error;
1463         IMGPROC_PRINT((
1464             IMGPROC_WARN(unable to unlock image processing plug-in mutex))
1465             harness_syserr());
1466         return -1;
1467     }
1468
1469     IMGPROC_PRINT((
1470         IMGPROC_INFO(image %s successfully sent to %s) ,name
1471         ,imgproc_data.successor))
1472 }
1473 /* if there is no successor plug-in, store the processed image */
1474 else
1475 {
1476     /* construct the whole path of the image file */
1477     storename = (char*)malloc( sizeof(char) *
1478                               (strlen(imgproc_data.targetdir) +2));
1479     if (storename == NULL)
1480     {
1481         IMGPROC_PRINT((
1482             IMGPROC_WARN(allocating of memory failed))
1483             return -1;
1484     }
1485     strcpy(storename, imgproc_data.targetdir);
1486     strcat(storename, "/" );
1487
1488     tmpfile = storename;
1489
1490     /* put image file name and path together */
1491     storename = (char*)malloc( sizeof(char) * (strlen(tmpfile) +
1492                                               strlen(name) +1 ));
1493     if (storename == NULL)
1494     {
1495         IMGPROC_PRINT((
1496             IMGPROC_WARN(allocating of memory failed))
1497             return -1;
1498     }
1499     strcpy( storename, tmpfile);
1500     strcat( storename, name);
1501
1502     /* store the image */
1503     imgproc_write_image( blob, size, storename);
1504
1505     IMGPROC_PRINT((
1506         IMGPROC_INFO(image %s successfully stored),name))
1507
1508     /* this section was used to simulate a broken pipeline and calling the
1509        PPM for restoration */
1510     /*
1511     if (testswitch == 0)
1512     {
1513         imgproc_createremoteref( &remoteobj, "1003",imgproc_data.ppmnode);
1514         rmixppmclient_repairpipe ( remoteobj,imgproc_data.successor);
1515         rmix_remoteref_destroy( &remoteobj);
1516         testswitch = 1;
1517     }
1518     */
```

A. Appendix

```
1520     /* prepare the remote reference for the predecessor */
1521     ret = imgproc_createremoteref( &remoteobj, "1002",
1522                                   imgproc_data.predecessor);
1523     if (ret != 0)
1524     {
1525         IMGPROC_PRINT((
1526             IMGPROC_WARN(could not create remote object references)))
1527
1528         /* prepare the remote reference for the PPM node */
1529         ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
1530                                       imgproc_data.ppmnode);
1531         if (ret != 0)
1532         {
1533             IMGPROC_PRINT((
1534                 IMGPROC_WARN(could not create remote object references)))
1535             return -1;
1536         }
1537
1538         /* call the PPM for pipeline restoration */
1539         ret = rmixppmclient_repairpipe( remoteobj_ppm,
1540                                       imgproc_data.predecessor);
1541         if (ret != 0)
1542         {
1543             IMGPROC_PRINT((
1544                 IMGPROC_WARN(could not repair pipe)))
1545             rmix_remoteref_destroy( &remoteobj_ppm);
1546             return -1;
1547         }
1548
1549         rmix_remoteref_destroy( &remoteobj_ppm);
1550         ret = imgproc_createremoteref( &remoteobj, "1002",
1551                                       imgproc_data.predecessor);
1552         if (ret != 0)
1553         {
1554             IMGPROC_PRINT((
1555                 IMGPROC_WARN(could not create remote object
1556                             references)))
1557             return -1;
1558         }
1559     }
1560
1561     /* inform the predecessor about a successfully stored image */
1562     ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
1563     if (ret != 0)
1564     {
1565
1566         IMGPROC_PRINT((
1567             IMGPROC_WARN(could not call remote object for passing image)))
1568
1569         /* prepare the remote reference of the PPM node */
1570         ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
1571                                       imgproc_data.ppmnode);
1572         if (ret != 0)
1573         {
1574             IMGPROC_PRINT((
1575                 IMGPROC_WARN(could not create remote object references)))
1576             return -1;
1577         }
1578
1579         /* call the PPM for pipeline restoration */
1580         ret = rmixppmclient_repairpipe( remoteobj_ppm,
1581                                       imgproc_data.predecessor);
```

A. Appendix

```
1582     if (ret != 0)
1583     {
1584         IMGPROC_PRINT((
1585             IMGPROC_WARN(could not repair pipe)))
1586         rmix_remoteref_destroy( &remoteobj_ppm);
1587         return -1;
1588     }
1589
1590     rmix_remoteref_destroy( &remoteobj_ppm);
1591     ret = imgproc_createremoteref( &remoteobj, "1002",
1592                                   imgproc_data.predecessor);
1593
1594     if (ret != 0)
1595     {
1596         IMGPROC_PRINT((
1597             IMGPROC_WARN(could not create remote object
1598                           references)))
1599         return -1;
1600     }
1601
1602     /* resend the message */
1603     ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
1604     if (ret != 0)
1605     {
1606         IMGPROC_PRINT((
1607             IMGPROC_WARN(could not call remote object for passing
1608                           image)))
1609         return -1;
1610     }
1611
1612     /* destroy the remote reference */
1613     ret = rmix_remoteref_destroy( &remoteobj);
1614     if (ret != 0)
1615     {
1616         IMGPROC_PRINT((
1617             IMGPROC_WARN(could not destroy remote object references)))
1618     }
1619
1620     FREE(tmpfile);
1621     FREE(storename);
1622
1623     /* Lock image processing plug-in mutex. */
1624     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
1625         errno = error;
1626         IMGPROC_PRINT((
1627             IMGPROC_WARN(unable to lock image processing plug-in mutex)))
1628         harness_syserr();
1629         return -1;
1630     }
1631
1632     /* store the image if it was not stored yet */
1633     ret = imgproc_worklist_find_elementposition( name, worklist);
1634     if (ret == -1)
1635     {
1636         imgproc_worklist_insert( name, 0, NULL, &worklist);
1637
1638         /* one more image was processed, decrease the counter */
1639         imgproc_decrease_imagecounter();
1640     }
1641
1642     /* Unlock image processing plug-in mutex. */
1643     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
1644     {
```

A. Appendix

```
1646         errno = error;
1647         IMGPROC_PRINT((
1648             IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
1649         harness_syserr();
1650         return -1;
1651     }
1652
1653     IMGPROC_PRINT((
1654         IMGPROC_INFO(acknowledgment for %s successfully sent to %s) ,name
1655         ,imgproc_data.predecessor))
1656     }
1657
1658     /* check if the iamge counter is zero and all images were processed */
1659     if (imgproc_data.imagecounter == 0)
1660     {
1661         /* if yes shutdown the plug-in and the Harness kernel */
1662         harness_kernel_shutdown();
1663     }
1664     return(0);
1665 }
1666
1667 /*
1668  * Passes an image to the next plug-in (client-side method stub).
1669  *
1670  * \param *remoteobj  ID of the remote object
1671  * \param *name       Name of the image file
1672  * \param *blob       Image data
1673  * \param size       Size of the image data
1674  * \return           0 on success or -1 on any error
1675  */
1676 #undef __FUNC__
1677 #define __FUNC__ "rmiximgprocclient_passimage_oneway"
1678 int rmiximgprocclient_passimage_oneway ( rmix_remoteref_t *remoteobj ,
1679     char *name,
1680     unsigned char *blob ,
1681     unsigned int size)
1682 {
1683     const void *inary[3];
1684
1685     /* pack data */
1686     inary[0] = name;
1687     inary[1] = &size;
1688     inary[2] = blob;
1689
1690     /* Invoke remote object method by using a one way invocation. */
1691     if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface ,
1692         RMIXIMGPROC_METHODS_PASSIMAGE_INDEX, inary , 3))
1693     {
1694         int errno2 = errno;
1695         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
1696         errno = errno2;
1697         return -1;
1698     }
1699
1700     return 0;
1701 }
1702
1703 /*
1704  * Accepts acknowledgment calls that an image was successfully stored.
1705  * (server-side stub)
1706  */
```

A. Appendix

```
1708 *
1709 * \param object The local object.
1710 * \param outary The output values array. The values array and its
1711 * containing values are allocated before the call with the
1712 * exception of variable arrays and strings. They are allocated
1713 * dynamically or explicitly set to NULL by this function. In
1714 * the case of variable arrays, the length value is allocated
1715 * before the call and is set to 0 if its variable array is
1716 * NULL.
1717 * \param outcnt The output values count.
1718 * \param inary The input values array. The values array and its containing
1719 * values are not modified or deallocated by this call.
1720 * \param incnt The input values count.
1721 * \return 0 on success or -1 on any error.
1722 */
1723 #undef __FUNC__
1724 #define __FUNC__ "rmiximgproc_imageprocessed_call"
1725 RMIX_METHOD_CALL(rmiximgproc_imageprocessed_call)
1726 {
1727     int result = 0;
1728     char *imgname; /* image name */
1729
1730 #ifdef DEBUG
1731     /* Check object parameter. */
1732     /*
1733     if (NULL == object)
1734     {
1735         errno = EINVAL;
1736         RMIX_LOG((RMIX_WARN(object parameter is null)))
1737         errno = EINVAL;
1738         return -1;
1739     }
1740     */
1741     /* Check outary parameter. */
1742     if ((NULL == outary)&&(0 != outcnt))
1743     {
1744         errno = EINVAL;
1745         RMIX_LOG((RMIX_WARN(outary parameter is null)))
1746         errno = EINVAL;
1747         return -1;
1748     }
1749
1750     /* Check outcnt parameter. */
1751     if ((0 == outcnt)&&(NULL != outary))
1752     {
1753         errno = EINVAL;
1754         RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
1755         errno = EINVAL;
1756         return -1;
1757     }
1758
1759     /* Check inary parameter. */
1760     if ((NULL == inary)&&(0 != incnt))
1761     {
1762         errno = EINVAL;
1763         RMIX_LOG((RMIX_WARN(inary parameter is null)))
1764         errno = EINVAL;
1765         return -1;
1766     }
1767
1768     /* Check incnt parameter. */
1769     if ((0 == incnt)&&(NULL != inary))
```

A. Appendix

```
1772     {
1773         errno = EINVAL;
1774         RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
1775         errno = EINVAL;
1776         return -1;
1777     }
1778 #endif /* DEBUG */
1779
1780     /* Prepare input. */
1781     imgname = (char*)inary[0];
1782
1783     /* Call method function. */
1784     result = rmiximgproc_imageprocessed( imgname);
1785
1786     /* Prepare output. */
1787     *(int*)(outary[0]) = result;
1788     /* Reset errno if needed. */
1789     if (0 != result)
1790     {
1791         errno = 0;
1792     }
1793     return 0;
1794 }
1795
1796 /*
1797  * Deletes a processed image from the internal worklist. If all images are
1798  * processed the plug-in is shutdown. If there is a predecessor plug-in unit,
1799  * the message is forwarded to it.
1800  *
1801  * \param  *name Name of the image.
1802  * \return      0 on success or -1 on any error.
1803  */
1804 #undef __FUNC__
1805 #define __FUNC__ "rmiximgproc_imageprocessed"
1806 int rmiximgproc_imageprocessed (char* name)
1807 {
1808     int error;
1809     int ret;
1810     rmix_remoteref_t *remoteobj;
1811     rmix_remoteref_t *remoteobj_ppm;
1812
1813     /* if a predecessor plug-in exists, forward the name to it */
1814     if (strcmp(imgproc_data.predecessor, "nada") != 0)
1815     {
1816         /* prepare the remote reference of the predecessor */
1817         ret = imgproc_createremoteref( &remoteobj, "1002",
1818                                     imgproc_data.predecessor);
1819         if (ret != 0)
1820         {
1821             IMGPROC_PRINT((
1822                 IMGPROC_WARN(could not create remote object references)))
1823
1824             /* prepare the remote reference of the PPM node */
1825             ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
1826                                         imgproc_data.ppmnode);
1827             if (ret != 0)
1828             {
1829                 IMGPROC_PRINT((
1830                     IMGPROC_WARN(could not create remote object references)))
1831                 return -1;
1832             }
1833         }
1834     }
1835 }
```

A. Appendix

```
1834     /* call the PPM for pipeline restoration */
1835     ret = rmixppmclient_repairpipe( remoteobj_ppm,
1836                                   imgproc_data.predecessor);
1837
1838     if (ret != 0)
1839     {
1840         IMGPROC_PRINT((
1841             IMGPROC_WARN(could not repair pipe)))
1842         rmix_remoteref_destroy( &remoteobj_ppm);
1843         return -1;
1844     }
1845
1846     rmix_remoteref_destroy( &remoteobj_ppm);
1847     ret = imgproc_createremoteref( &remoteobj, "1002",
1848                                   imgproc_data.predecessor);
1849
1850     if (ret != 0)
1851     {
1852         IMGPROC_PRINT((
1853             IMGPROC_WARN(could not create remote object
1854                         references)))
1855         return -1;
1856     }
1857 }
1858
1859 /* pass the acknowledgment with an onway RPC call */
1860 ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
1861 if (ret != 0)
1862 {
1863     IMGPROC_PRINT((
1864         IMGPROC_WARN(could not call remote object for passing image)))
1865 }
1866
1867 /* prepare the remote reference of the PPM call */
1868 ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
1869                               imgproc_data.ppmnode);
1870
1871 if (ret != 0)
1872 {
1873     IMGPROC_PRINT((
1874         IMGPROC_WARN(could not create remote object references)))
1875     return -1;
1876 }
1877
1878 /* call the PPM for pipeline restoration */
1879 ret = rmixppmclient_repairpipe( remoteobj_ppm,
1880                               imgproc_data.predecessor);
1881
1882 if (ret != 0)
1883 {
1884     IMGPROC_PRINT((
1885         IMGPROC_WARN(could not repair pipe)))
1886     rmix_remoteref_destroy( &remoteobj_ppm);
1887     return -1;
1888 }
1889
1890 rmix_remoteref_destroy( &remoteobj_ppm);
1891 ret = imgproc_createremoteref( &remoteobj, "1002",
1892                               imgproc_data.predecessor);
1893
1894 if (ret != 0)
1895 {
1896     IMGPROC_PRINT((
1897         IMGPROC_WARN(could not create remote object
1898                     references)))
1899     return -1;
1900 }
1901
1902 /* resend the message */
```


A. Appendix

```
1898     ret = rmixmapprocclient_imageprocessed_oneway ( remoteobj, name);
1899     if (ret != 0)
1900     {
1901         IMGPROC_PRINT((
1902             IMGPROC_WARN(could not call remote object for passing
1903                 image)))
1904         return -1;
1905     }
1906
1907     /* destroy the remote reference */
1908     ret = rmixmap_remoteref_destroy( &remoteobj);
1909     if (ret != 0)
1910     {
1911         IMGPROC_PRINT((
1912             IMGPROC_WARN(could not destroy remote object references)))
1913     }
1914
1915     /* delete the image from the internal backup list */
1916     /* Lock image processing plug-in mutex. */
1917     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
1918         errno = error;
1919         IMGPROC_PRINT((
1920             IMGPROC_WARN(unable to lock image processing plug-in mutex)))
1921         harness_syserr();
1922         return -1;
1923     }
1924
1925     ret = imgproc_worklist_find_elementposition( name, worklist);
1926     if (ret != -1)
1927     {
1928         imgproc_worklist_delete( name, &worklist);
1929         /* one more image was processed, decrease the counter */
1930         imgproc_decrease_imagecounter();
1931     }
1932
1933     /* Unlock image processing plug-in mutex. */
1934     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
1935     {
1936         errno = error;
1937         IMGPROC_PRINT((
1938             IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
1939         harness_syserr();
1940         return -1;
1941     }
1942
1943     IMGPROC_PRINT((
1944         IMGPROC_INFO(acknowledgement for %s successfully sent to %s) ,name
1945             ,imgproc_data.predecessor))
1946 }
1947 else
1948 {
1949     /* if it is the first pipeline unit, only delete it from the backup
1950     list */
1951
1952     /* Lock image processing plug-in mutex. */
1953     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
1954         errno = error;
1955         IMGPROC_PRINT((
1956             IMGPROC_WARN(unable to lock image processing plug-in mutex)))
1957         harness_syserr();
1958         return -1;
```

A. Appendix

```
1960     }
1962     ret = imgproc_worklist_find_elementposition( name, worklist);
1964     if (ret != -1)
1966     {
1968         imgproc_worklist_delete( name, &worklist);
1970         /* one more image was processed, decrease the counter */
1972         imgproc_decrease_imagecounter();
1974     }
1976     /* Unlock image processing plug-in mutex. */
1978     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
1980     {
1982         errno = error;
1984         IMGPROC_PRINT((
1986             IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
1988         harness_syserr();
1990         return -1;
1992     }
1994     /* check if the counter is zero and all images were processed */
1996     if (imgproc_data.imagecounter == 0)
1998     {
2000         /* if yes shutdown the plug-in and the Harness kernel */
2002         harness_kernel_shutdown();
2004     }
2006     return 0;
2008 }
2010
2012 /*
2014 * Sends an acknowledgment for a successfully stored iamge (client-side
2016 * method stub).
2018 *
2020 * \param *remoteobj ID of the remote object
2022 * \param *name      Name of the image file
2024 * \return          0 on success or -1 on any error
2026 */
2028 #undef __FUNC__
2030 #define __FUNC__ "rmiximgprocclient_imageprocessed_oneway"
2032 int rmiximgprocclient_imageprocessed_oneway ( rmix_remoteref_t *remoteobj,
2034                                             char *name)
2036 {
2038     const void *inary[1];
2040     /* pack data */
2042     inary[0] = name;
2044     /* Invoke remote object method by using a one way invocation. */
2046     if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface,
2048                          RMIXIMGPROC_METHODS_IMAGEPROCESSED_INDEX, inary, 1))
2050     {
2052         int errno2 = errno;
2054         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
2056         errno = errno2;
2058         return -1;
2060     }
2062     return 0;
2064 }
2066 }
```

A. Appendix

```
2024 /*
2025  * Accepts calls for updating the predecessor entry of the plug-in.
2026  * (server-side stub)
2027  *
2028  * \param object The local object.
2029  * \param outary The output values array. The values array and its
2030  * containing values are allocated before the call with the
2031  * exception of variable arrays and strings. They are allocated
2032  * dynamically or explicitly set to NULL by this function. In
2033  * the case of variable arrays, the length value is allocated
2034  * before the call and is set to 0 if its variable array is
2035  * NULL.
2036  * \param outcnt The output values count.
2037  * \param inary The input values array. The values array and its containing
2038  * values are not modified or deallocated by this call.
2039  * \param incnt The input values count.
2040  * \return 0 on success or -1 on any error.
2041  */
2042 #undef __FUNC__
2043 #define __FUNC__ "rmiximgproc_updatesuccessor_call"
2044 RMX_METHOD_CALL(rmiximgproc_updatesuccessor_call)
2045 {
2046     int result = 0;
2047
2048     char *predecessor; /* successor name */
2049
2050 #ifdef DEBUG
2051     /* Check object parameter. */
2052     /*
2053     if (NULL == object)
2054     {
2055         errno = EINVAL;
2056         RMX_LOG((RMX_WARN(object parameter is null)))
2057         errno = EINVAL;
2058         return -1;
2059     }
2060     */
2061     /* Check outary parameter. */
2062     if ((NULL == outary)&&(0 != outcnt))
2063     {
2064         errno = EINVAL;
2065         RMX_LOG((RMX_WARN(outary parameter is null)))
2066         errno = EINVAL;
2067         return -1;
2068     }
2069
2070     /* Check outcnt parameter. */
2071     if ((0 == outcnt)&&(NULL != outary))
2072     {
2073         errno = EINVAL;
2074         RMX_LOG((RMX_WARN(outcnt parameter is zero)))
2075         errno = EINVAL;
2076         return -1;
2077     }
2078
2079     /* Check inary parameter. */
2080     if ((NULL == inary)&&(0 != incnt))
2081     {
2082         errno = EINVAL;
2083         RMX_LOG((RMX_WARN(inary parameter is null)))
2084         errno = EINVAL;
2085         return -1;
2086     }
2087 }
```

A. Appendix

```
2086     }
2088     /* Check incnt parameter. */
2089     if ((0 == incnt)&&(NULL != inary))
2090     {
2091         errno = EINVAL;
2092         RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
2093         errno = EINVAL;
2094         return -1;
2095     }
2096 #endif /* DEBUG */
2098     /* Prepare input. */
2099     predecessor = (char*)inary[0];
2100
2101     /* Call method function. */
2102     result = rmiximgproc_updatesuccessor( predecessor);
2103
2104     /* Prepare output. */
2105     *(int*) (outary[0]) = result;
2106     /* Reset errno if needed. */
2107     if (0 != result)
2108     {
2109         errno = 0;
2110     }
2111     return 0;
2112 }
2113
2114 /*
2115  * Updates the predecessor entry of the plug-in.
2116  *
2117  * \param  *predecessor  Name of the predecessor plug-in.
2118  * \return          0 on success or -1 on any error.
2119  */
2120 #undef __FUNC__
2121 #define __FUNC__ "rmiximgproc_updatesuccessor"
2122 int rmiximgproc_updatesuccessor (char *predecessor)
2123 {
2124     /* delete the old entry */
2125     FREE(imgproc_data.predecessor);
2126
2127     /* allocate memory for the new entry */
2128     imgproc_data.predecessor = (char*)malloc( sizeof(char) *
2129                                             (strlen(predecessor) + 1));
2130     if (imgproc_data.predecessor == NULL)
2131     {
2132         IMGPROC_PRINT((
2133             IMGPROC_WARN(could not allocate memory)))
2134         return -1;
2135     }
2136     /* store the new entry */
2137     strcpy( imgproc_data.predecessor , predecessor);
2138
2139     return 0;
2140 }
2141
2142
2143 /*
2144  * Accepts call for updating the successor entry of the plug-in and returns the
2145  * current image counter. (server-side stub)
2146  *
2147  * \param  object  The local object.
```

A. Appendix

```
2150 * \param outary The output values array. The values array and its
2151 * containing values are allocated before the call with the
2152 * exception of variable arrays and strings. They are allocated
2153 * dynamically or explicitly set to NULL by this function. In
2154 * the case of variable arrays, the length value is allocated
2155 * before the call and is set to 0 if its variable array is
2156 * NULL.
2157 * \param outcnt The output values count.
2158 * \param inary The input values array. The values array and its containing
2159 * values are not modified or deallocated by this call.
2160 * \param incnt The input values count.
2161 * \return 0 on success or -1 on any error.
2162 */
2163 #undef __FUNC__
2164 #define __FUNC__ "rmiximgproc_updatepredecessor_call"
2165 RMIX_METHOD_CALL(rmiximgproc_updatepredecessor_call)
2166 {
2167     int result = 0;
2168
2169     char *successor; /* successor name */
2170     unsigned int counter; /* current image counter */
2171
2172 #ifdef DEBUG
2173     /* Check object parameter. */
2174     /*
2175     if (NULL == object)
2176     {
2177         errno = EINVAL;
2178         RMIX_LOG((RMIX_WARN(object parameter is null)))
2179         errno = EINVAL;
2180         return -1;
2181     }
2182     */
2183     /* Check outary parameter. */
2184     if ((NULL == outary)&&(0 != outcnt))
2185     {
2186         errno = EINVAL;
2187         RMIX_LOG((RMIX_WARN(outary parameter is null)))
2188         errno = EINVAL;
2189         return -1;
2190     }
2191
2192     /* Check outcnt parameter. */
2193     if ((0 == outcnt)&&(NULL != outary))
2194     {
2195         errno = EINVAL;
2196         RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
2197         errno = EINVAL;
2198         return -1;
2199     }
2200
2201     /* Check inary parameter. */
2202     if ((NULL == inary)&&(0 != incnt))
2203     {
2204         errno = EINVAL;
2205         RMIX_LOG((RMIX_WARN(inary parameter is null)))
2206         errno = EINVAL;
2207         return -1;
2208     }
2209
2210     /* Check incnt parameter. */
2211     if ((0 == incnt)&&(NULL != inary))
2212     {
```

A. Appendix

```
2212     errno = EINVAL;
2213     RMX_LOG((RMX_WARN(incnt paramet is zero)))
2214     errno = EINVAL;
2215     return -1;
2216 }
2217 #endif /* DEBUG */
2218
2219     /* Prepare input. */
2220     successor = (char*)inary[0];
2221
2222     /* Call method function. */
2223     result = rmiximgproc_updatepredecessor( successor , &counter);
2224
2225     /* Prepare output. */
2226     *(int*) (outary[0]) = result;
2227     *(unsigned int*)(outary[1]) = counter;
2228     /* Reset errno if needed. */
2229     if (0 != result)
2230     {
2231         errno = 0;
2232     }
2233
2234     return 0;
2235 }
2236
2237
2238 /*
2239 * Updates the successor entry of the plug-in and returns the current image
2240 * counter.
2241 *
2242 * \param *successor Name of the successor plug-in.
2243 * \param *counter Current image counter.
2244 * \return 0 on success or -1 on any error.
2245 */
2246 #undef __FUNC__
2247 #define __FUNC__ "rmiximgproc_updatepredecessor"
2248 int rmiximgproc_updatepredecessor( char *successor ,
2249                                   unsigned int *counter)
2250 {
2251     /* delete the old entry */
2252     FREE(imgproc_data.successor);
2253
2254     /* allocate memory for the new entry */
2255     imgproc_data.successor = (char*)malloc( sizeof(char) * (strlen(successor)
2256                                                         + 1));
2257     if (imgproc_data.successor == NULL)
2258     {
2259         IMGPROC_PRINT((
2260             IMGPROC_WARN(could not allocate memory)))
2261         return -1;
2262     }
2263     /* store the new entry */
2264     strcpy( imgproc_data.successor , successor);
2265
2266     /* return the image counter */
2267     *counter = imgproc_data.imagecounter;
2268
2269     return 0;
2270 }
2271
2272
2273 /*
2274 * Accepts calls for sending the images in the backup list again to the
```

A. Appendix

```

    * successor (server-side stub).
2276 *
    * \param object The local object.
2278 * \param outary The output values array. The values array and its
    * containing values are allocated before the call with the
2280 * exception of variable arrays and strings. They are allocated
    * dynamically or explicitly set to NULL by this function. In
2282 * the case of variable arrays, the length value is allocated
    * before the call and is set to 0 if its variable array is
2284 * NULL.
    * \param outcnt The output values count.
2286 * \param inary The input values array. The values array and its containing
    * values are not modified or deallocated by this call.
2288 * \param incnt The input values count.
    * \return 0 on success or -1 on any error.
2290 */
#undef __FUNC__
2292 #define __FUNC__ "rmiximgproc_sendworklist_call"
    RMIX_METHOD_CALL(rmiximgproc_sendworklist_call)
2294 {
    int result = 0;
2296
    #ifdef DEBUG
2298 /* Check object parameter. */
    /*
2300 if (NULL == object)
    {
2302     errno = EINVAL;
    RMIX_LOG((RMIX_WARN(object parameter is null)))
2304     errno = EINVAL;
    return -1;
2306 }
    */
2308 /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
2310 {
    errno = EINVAL;
    RMIX_LOG((RMIX_WARN(outary parameter is null)))
2312     errno = EINVAL;
    return -1;
2314 }
2316
    /* Check outcnt parameter. */
2318 if ((0 == outcnt)&&(NULL != outary))
    {
2320     errno = EINVAL;
    RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
2322     errno = EINVAL;
    return -1;
2324 }
2326
    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
2328 {
    errno = EINVAL;
    RMIX_LOG((RMIX_WARN(inary parameter is null)))
2330     errno = EINVAL;
    return -1;
2332 }
2334
    /* Check incnt parameter. */
2336 if ((0 == incnt)&&(NULL != inary))
    {
```

A. Appendix

```
2338     errno = EINVAL;
2339     RMX_LOG((RMX_WARN(incont paramet is zero)))
2340     errno = EINVAL;
2341     return -1;
2342 }
2343 #endif /* DEBUG */
2344
2345     /* Call method function. */
2346     result = rmixmapproc_sendworklist();
2347
2348     /* Prepare output. */
2349     *(int*)(outary[0]) = result;
2350     /* Reset errno if needed. */
2351     if (0 != result)
2352     {
2353         errno = 0;
2354     }
2355     return 0;
2356 }
2357
2358 /*
2359 * Sends again the images in the worklist to the successor plug-in.
2360 *
2361 * \return 0 on success or -1 on any error.
2362 */
2363 #undef __FUNC__
2364 #define __FUNC__ "rmixmapproc_sendworklist"
2365 int rmixmapproc_sendworklist()
2366 {
2367     int error;
2368     int ret;
2369     worklist_t *tmpworklist;
2370     worklist_t *worklistcopy = NULL;
2371     rmixmap_remoteref_t *remoteobj;
2372
2373     /* avoid that other threads change the backup list */
2374
2375     /* Lock image processing plug-in mutex. */
2376     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex)))
2377     {
2378         errno = error;
2379         IMGPROC_PRINT((
2380             IMGPROC_WARN(unable to lock image processing plug-in mutex)))
2381         harness_syserr();
2382         return -1;
2383     }
2384
2385     /* copy liste */
2386
2387     tmpworklist = worklist;
2388     while(tmpworklist != NULL)
2389     {
2390
2391         imgproc_worklist_insert( tmpworklist->name, tmpworklist->size,
2392                                 tmpworklist->blob, &worklistcopy);
2393
2394
2395         tmpworklist = tmpworklist->next;
2396     }
2397
2398     /* Unlock image processing plug-in mutex. */
2399     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
```


A. Appendix

```
2402     {
2403         errno = error;
2404         IMGPROC_PRINT((
2405             IMGPROC_WARN(unable to unlock image processing plug-in mutex))
2406             harness_syserr());
2407         return -1;
2408     }
2409
2410     /* debug print out */
2411     imgproc_worklist_printlist(worklistcopy);
2412
2413     /* resend the images but do not put them again in the list */
2414     tmpworklist = worklistcopy;
2415
2416     /* prepare the remote reference of the successor */
2417     ret = imgproc_createremoteref( &remoteobj, "1002", imgproc_data.successor);
2418     if (ret != 0)
2419     {
2420         IMGPROC_PRINT((
2421             IMGPROC_WARN(could not create remote object))
2422             return -1;
2423     }
2424
2425     while(tmpworklist != NULL)
2426     {
2427         /* forward the image */
2428         IMGPROC_PRINT((
2429             IMGPROC_INFO(resending image %s), tmpworklist->name))
2430         ret = rmixmapclient_passimage_oneway ( remoteobj, tmpworklist->name,
2431             tmpworklist->blob,
2432             tmpworklist->size);
2433
2434         if (ret != 0)
2435         {
2436             IMGPROC_PRINT((
2437                 IMGPROC_WARN(could not send image)))
2438             tmpworklist = tmpworklist->next;
2439             continue;
2440         }
2441         tmpworklist = tmpworklist->next;
2442     }
2443
2444     /* destroy the remote reference */
2445     ret = rmixmapremoteref_destroy( &remoteobj);
2446     if (ret != 0)
2447     {
2448         IMGPROC_PRINT((
2449             IMGPROC_WARN(could not destroy remote object references))
2450         }
2451
2452     /* delete the copy of the backup list */
2453     imgproc_worklist_destroylist( &worklistcopy);
2454
2455     return 0;
2456 }
2457
2458 /*
2459 * Call the Parallel Plug-in Manager for repairing the pipeline (client-side
2460 * method stub).
2461 *
2462 * \param *remoteobj ID of the remote object
2463 * \param *node Name of the not reachable node
2464 * \return 0 on success or -1 on any error

```

A. Appendix

```
2464 */
2465 #undef __FUNC__
2466 #define __FUNC__ "rmixppmclient_repairpipe"
2467 int rmixppmclient_repairpipe ( rmix_remoteref_t *remoteobj,
2468                               char *node)
2469 {
2470     const void *inary[1];
2471     void *outary[1];
2472     int result;
2473
2474     /* Prepare parameters. */
2475     inary[0] = node;
2476     outary[0] = &result;
2477
2478     /* Invoke remote object method. */
2479     if (0 != rmix_invoke(remoteobj, &rmixppmclient_interface,
2480                          RMIXPPM_METHODS_REPAIRPIPE_INDEX, outary, 1,
2481                          inary, 1))
2482     {
2483         int errno2 = errno;
2484         RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
2485         errno = errno2;
2486         return -1;
2487     }
2488     return result;
2489 }
2490
2491
2492 /*
2493 * Creates a remote reference specified by objectid and node. The RPC protocol
2494 * is used.
2495 *
2496 * \param **remoteobj ID of the remote object
2497 * \param *objectid ID of the exported object
2498 * \param *node Name of the node
2499 * \return 0 on success or -1 on any error
2500 */
2501 #undef __FUNC__
2502 #define __FUNC__ "imgproc_createremoteref"
2503 int imgproc_createremoteref( rmix_remoteref_t **remoteobj,
2504                              char *objectid,
2505                              char *node)
2506 {
2507     int ret;
2508     char *remoteparameters;
2509
2510     /* prepare parameter for creating remote reference */
2511     remoteparameters = (char*)malloc( sizeof(char) *
2512                                       (strlen("PROTOCOL=RPC OBJECTID=%s HOST=%s") +
2513                                       strlen(objectid) +
2514                                       strlen(node) - 3 ));
2515
2516     if (remoteparameters == NULL)
2517     {
2518         IMGPROC_PRINT((
2519             IMGPROC_WARN(could not allocate memory)))
2520         return -1;
2521     }
2522     sprintf( remoteparameters, "PROTOCOL=RPC OBJECTID=%s HOST=%s",
2523             objectid, node);
2524
2525     /* create the remote reference */
2526     ret = rmix_remoteref_create6( &(*remoteobj), remoteparameters);
```

A. Appendix

```
2528     if (ret != 0)
2529     {
2530         IMGPROC_PRINT((
2531             IMGPROC_WARN(could not create remote object references)))
2532         return -1;
2533     }
2534     FREE(remoteparameters);
2535     return 0;
2536 }
2537
2538 /*****
2539 *
2540 * next section contains functions for processing the images
2541 *
2542 *****/
2543
2544 /*
2545 * Decreases the image counter by 1.
2546 *
2547 * \return 0 on success or -1 on any error
2548 */
2549 #undef __FUNC__
2550 #define __FUNC__ "imgproc_decrease_imagecounter"
2551 int imgproc_decrease_imagecounter()
2552 {
2553     /* decrease image counter */
2554     imgproc_data.imagecounter--;
2555     return 0;
2556 }
2557
2558
2559 /*
2560 * Rotates an image by 90 degrees.
2561 *
2562 * \param **blob Char array containing the image data.
2563 * \param *size Size of the char array.
2564 * \return 0 on success or -1 on any error
2565 */
2566 #undef __FUNC__
2567 #define __FUNC__ "imgproc_filter_rotation"
2568 int imgproc_filter_rotation( unsigned char **blob,
2569                             size_t *size)
2570 {
2571     MagickBooleanType status;
2572     MagickWand *magick_wand;
2573     PixelWand *background_color;
2574
2575     magick_wand = NewMagickWand();
2576     background_color = NewPixelWand();
2577
2578     /* create image from blob */
2579     status = MagickReadImageBlob( magick_wand, *blob, *size);
2580     if (status == MagickFalse)
2581     {
2582         ThrowWandException(magick_wand);
2583         return -1;
2584     }
2585
2586     /* get the background color, which is needed to fill empty triangles left
2587        over from rotating the image */
```

A. Appendix

```
2590 status = MagickGetImageBackgroundColor( magick_wand, background_color);
2591 if (status == MagickFalse)
2592 {
2593     ThrowWandException(magick_wand);
2594     return -1;
2595 }
2596
2597 /* rotate image by 90 degree */
2598 status = MagickRotateImage( magick_wand, background_color, 90);
2599 if (status == MagickFalse)
2600 {
2601     ThrowWandException(magick_wand);
2602     return -1;
2603 }
2604
2605 /* recreate blob */
2606 *blob = MagickGetImageBlob( magick_wand, size);
2607
2608 DestroyMagickWand(magick_wand);
2609
2610 return 0;
2611 }
2612
2613 /*
2614  * Puts an oilpainting filter on an image.
2615  *
2616  * \param **blob Char array containing the image data.
2617  * \param *size Size of the char array.
2618  * \return      0 on success or -1 on any error
2619  */
2620 #undef __FUNC__
2621 #define __FUNC__ "imgproc_filter_oilpainting"
2622 int imgproc_filter_oilpainting( unsigned char **blob,
2623                                size_t *size)
2624 {
2625     MagickBooleanType status;
2626     MagickWand *magick_wand;
2627
2628     magick_wand = NewMagickWand();
2629
2630     /* create image from blob */
2631     status = MagickReadImageBlob( magick_wand, *blob, *size);
2632     if (status == MagickFalse)
2633     {
2634         ThrowWandException(magick_wand);
2635         return -1;
2636     }
2637
2638     status = MagickOilPaintImage(magick_wand, 5);
2639     if (status == MagickFalse)
2640     {
2641         ThrowWandException(magick_wand);
2642         return -1;
2643     }
2644
2645     /* recreate blob */
2646     *blob = MagickGetImageBlob( magick_wand, size);
2647
2648     DestroyMagickWand(magick_wand);
2649
2650     return 0;
2651 }
2652 }
```

A. Appendix

```
2654
/*
2656 * Swirls the center of an image.
*
2658 * \param **blob Char array containing the image data.
* \param *size Size of the char array.
2660 * \return      0 on success or -1 on any error
*/
2662 #undef __FUNC__
#define __FUNC__ "imgproc_filter_swirl"
2664 int imgproc_filter_swirl( unsigned char **blob,
                           size_t *size)
2666 {
    MagickBooleanType status;
2668    MagickWand *magick_wand;

2670    magick_wand = NewMagickWand();

2672    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, *blob, *size);
2674    if (status == MagickFalse)
    {
2676        ThrowWandException(magick_wand);
        return -1;
2678    }

2680    status = MagickSwirlImage(magick_wand, 180);
    if (status == MagickFalse)
2682    {
        ThrowWandException(magick_wand);
2684        return -1;
    }

2686    /* recreate blob */
2688    *blob = MagickGetImageBlob( magick_wand, size);

2690    DestroyMagickWand(magick_wand);

2692    return 0;
2694 }

2696 /*
* Negates an image.
2698 *
* \param **blob Char array containing the image data.
2700 * \param *size Size of the char array.
* \return      0 on success or -1 on any error
2702 */
2704 #undef __FUNC__
#define __FUNC__ "imgproc_filter_negate"
2706 int imgproc_filter_negate( unsigned char **blob,
                           size_t *size)
{
2708    MagickBooleanType status;
    MagickWand *magick_wand;

2710    magick_wand = NewMagickWand();

2712    /* create image from blob */
2714    status = MagickReadImageBlob( magick_wand, *blob, *size);
    if (status == MagickFalse)
```

A. Appendix

```
2716     {
2717         ThrowWandException(magick_wand);
2718         return -1;
2719     }
2720
2721     status = MagickNegateImage(magick_wand, MagickFalse);
2722     if (status == MagickFalse)
2723     {
2724         ThrowWandException(magick_wand);
2725         return -1;
2726     }
2727
2728     /* recreate blob */
2729     *blob = MagickGetImageBlob( magick_wand, size);
2730
2731     DestroyMagickWand(magick_wand);
2732
2733     return 0;
2734 }
2735
2736 /*
2737 * Puts a solarize filter on an image.
2738 *
2739 * \param **blob Char array containing the image data.
2740 * \param *size Size of the char array.
2741 * \return      0 on success or -1 on any error
2742 */
2743 #undef __FUNC__
2744 #define __FUNC__ "imgproc_filter_solarize"
2745 int imgproc_filter_solarize( unsigned char **blob,
2746                             size_t *size)
2747 {
2748     MagickBooleanType status;
2749     MagickWand *magick_wand;
2750
2751     magick_wand = NewMagickWand();
2752
2753     /* create image from blob */
2754     status = MagickReadImageBlob( magick_wand, *blob, *size);
2755     if (status == MagickFalse)
2756     {
2757         ThrowWandException(magick_wand);
2758         return -1;
2759     }
2760
2761     status = MagickSolarizeImage( magick_wand, 80);
2762     if (status == MagickFalse)
2763     {
2764         ThrowWandException(magick_wand);
2765         return -1;
2766     }
2767
2768     /* recreate blob */
2769     *blob = MagickGetImageBlob( magick_wand, size);
2770
2771     DestroyMagickWand(magick_wand);
2772
2773     return 0;
2774 }
2775
2776 /*
2777 */
```

A. Appendix

```

2780 * Stores an image.
2780 *
2780 * \param **blob Char array containing the image data.
2782 * \param *size Size of the char array.
2782 * \param *filename Name of the new image file.
2784 * \return 0 on success or -1 on any error
2784 */
2786 #undef __FUNC__
2786 #define __FUNC__ "imgproc_write_image"
2788 int imgproc_write_image( unsigned char *blob,
2788                          size_t size,
2789                          char *filename)
2790 {
2792     MagickBooleanType status;
2792     MagickWand *magick_wand;
2794
2794     magick_wand = NewMagickWand();
2796
2796     /* create image from blob */
2798     status = MagickReadImageBlob( magick_wand, blob, size);
2798     if (status == MagickFalse)
2800     {
2800         ThrowWandException(magick_wand);
2802         return -1;
2802     }
2804
2804     status = MagickWriteImages( magick_wand, filename, MagickTrue);
2806     if (status == MagickFalse)
2806     {
2808         ThrowWandException(magick_wand);
2808         return -1;
2810     }
2812
2812     DestroyMagickWand(magick_wand);
2814
2814     return 0;
2816 }
2818
2818 /*
2818 * Returns the number of files found in a directory.
2820 *
2820 * \param *char Directory name.
2822 * \return Number of files
2822 */
2824 #undef __FUNC__
2824 #define __FUNC__ "imgproc_get_files"
2826 int imgproc_get_files( char *dir)
2826 {
2828     int images = 0;
2830
2830     DIR* DirectoryPointer;
2830     struct dirent* dp;
2832
2832     /* open the directory stream */
2834     DirectoryPointer = opendir(dir);
2836
2836     /* count the numbers of files */
2836     for (dp = readdir(DirectoryPointer); dp!=0; dp=readdir(DirectoryPointer) )
2838     {
2838         if (strcmp(dp->d_name, ".")!=0 && strcmp(dp->d_name, "..")!=0)
2840         {
2840             images++;

```

A. Appendix

```
2842     }
2843 }
2844
2845 /* close the directory stream */
2846 closedir(DirectoryPointer);
2847
2848 IMGPROC_PRINT((
2849     IMGPROC_INFO(%d images found), images))
2850 return images;
2851 }
2852
2853
2854 /*
2855  * Tries to load the image files in the specified directory and additional
2856  * files.
2857  *
2858  * \param  *imgdir    Directory containing the image files.
2859  * \return          0 on success or -1 on any error.
2860  */
2861 #undef __FUNC__
2862 #define __FUNC__ "imgproc_loadimages"
2863 int imgproc_loadimages( char *imgdir)
2864 {
2865     int          images;
2866     int          ret;
2867     char         *imgfile;          /* path + filename */
2868     char         *imgname;         /* image name */
2869     char         *tmpfile;
2870
2871     rmix_remoteref_t *remoteobj;
2872
2873     DIR          *DirectoryPointer; /* directorystream object */
2874     struct dirent *dp;
2875
2876     imgfile = NULL;
2877     tmpfile = NULL;
2878
2879     /* get the number of image files */
2880     images = imgproc_get_files(imgdir);
2881
2882     if (images == 0)
2883     {
2884         IMGPROC_PRINT((
2885             IMGPROC_WARN(no images in source directory)))
2886         return -1;
2887     }
2888     else
2889     {
2890         /* set the image counter of all plug-in units by forwarding it*/
2891         if (strcmp(imgproc_data.successor, "nada") != 0)
2892         {
2893             /* set the parameters for creating the remote reference to
2894              the successor */
2895             ret = imgproc_createremoteref( &remoteobj, "1002",
2896                 imgproc_data.successor);
2897
2898             if (ret != 0)
2899             {
2900                 IMGPROC_PRINT((
2901                     IMGPROC_WARN(could not create remote object)))
2902                 return -1;
2903             }
2904
2905             /* forward the image counter */

```


A. Appendix

```
2906     ret = rmixmapprocclient_setimagecounter ( remoteobj, images);
2907     if (ret != 0)
2908     {
2909         IMGPROC_PRINT((
2910             IMGPROC_WARN(could not set image counter)))
2911         return -1;
2912     }
2913
2914     /* destroy the remote reference */
2915     ret = rmixmapremoteref_destroy( &remoteobj);
2916     if (ret != 0)
2917     {
2918         IMGPROC_PRINT((
2919             IMGPROC_WARN(could not destroy remote object references)))
2920     }
2921 }
2922 /* set the own image counter */
2923 imgproc_data.imagecounter = images;
2924
2925 /* start with processing the images */
2926
2927 /* open a directorystream */
2928 DirectoryPointer = opendir( imgdir);
2929 for (dp=readdir(DirectoryPointer); dp!=0; dp=readdir(DirectoryPointer))
2930 {
2931     if (strcmp(dp->d_name, ".")!=0 && strcmp(dp->d_name, "..")!=0)
2932     {
2933         IMGPROC_PRINT((
2934             IMGPROC_INFO(reading file %s), dp->d_name))
2935
2936         /* copy image name */
2937         imgname = (char*)malloc( sizeof(char) * (strlen(dp->d_name) +1));
2938         if (imgname == NULL)
2939         {
2940             IMGPROC_PRINT((
2941                 IMGPROC_WARN(allocating of memory failed)))
2942             return -1;
2943         }
2944         strcpy(imgname, dp->d_name);
2945
2946         /* construct the whole path of the image */
2947         imgfile = (char*)malloc( sizeof(char) * (strlen(imgdir) +2));
2948         if (imgfile == NULL)
2949         {
2950             IMGPROC_PRINT((
2951                 IMGPROC_WARN(allocating of memory failed)))
2952             return -1;
2953         }
2954         strcpy(imgfile, imgdir);
2955         strcat(imgfile, "/");
2956
2957         tmpfile = imgfile;
2958
2959         /* add the filename to the path */
2960         imgfile = (char*)malloc( sizeof(char) * (strlen(tmpfile) +
2961             strlen(dp->d_name) +1 ));
2962         if (imgfile == NULL)
2963         {
2964             IMGPROC_PRINT((
2965                 IMGPROC_WARN(allocating of memory failed)))
2966             return -1;
2967         }
2968     }
2969 }
```

A. Appendix

```
2968     strcpy( imgfile , tmpfile );
2969     strcat( imgfile , dp->d_name);
2970
2971     FREE(tmpfile);
2972
2973     IMGPROC_PRINT((
2974         IMGPROC_INFO(full filename is %s), imgfile))
2975
2976     /* process the image and forward it */
2977     ret = imgproc_create_magickwand(imgfile , imgname);
2978     if (ret != 0)
2979     {
2980         IMGPROC_PRINT((
2981             IMGPROC_WARN(cannot start processing the first filter)))
2982         FREE(imgname);
2983         FREE(imgfile);
2984         return -1;
2985     }
2986
2987     FREE(imgname);
2988     FREE(imgfile);
2989 }
2990     closedir(DirectoryPointer);
2991 }
2992
2993 /* check if the counter is zero and all images were processed */
2994 if (imgproc_data.imagecounter == 0)
2995 {
2996     /* if yes shutdown the plug-in and the Harness kernel */
2997     harness_kernel_shutdown();
2998 }
2999
3000 return 0;
3001 }
3002
3003
3004
3005 /*
3006 * Processes an image and forwards it to the next plug-in unit. This function
3007 * is only executed by the first unit in the pipeline.
3008 *
3009 * \param *imgfile File name of the image + path
3010 * \param *imgname File name
3011 * \return 0 on success or -1 on any error
3012 */
3013 #undef __FUNC__
3014 #define __FUNC__ "imgproc_create_magickwand"
3015 int imgproc_create_magickwand( char *imgfile , char *imgname)
3016 {
3017     MagickBooleanType status;
3018     MagickWand *magick_wand;
3019
3020     unsigned char *blob;
3021     size_t size;
3022     char *storename;
3023     char *tmpfile;
3024
3025     rmix_remoteref_t *remoteobj;
3026     rmix_remoteref_t *remoteobj_ppm;
3027
3028     int ret;
3029     int error;
3030 }
```

A. Appendix

```
magick_wand = NewMagickWand();
3032
/* Read an image. */
3034 status = MagickReadImage( magick_wand, imgfile);
if (status == MagickFalse)
3036 {
    ThrowWandException(magick_wand);
3038     return -1;
}
3040
/* create image blob */
3042 blob = MagickGetImageBlob( magick_wand, &size);
3044 DestroyMagickWand(magick_wand);
3046
/* use first filter */
ret = filters[imgproc_data.filter](&blob, &size);
3048 if (ret != 0)
{
    IMGPROC_PRINT((
3050         IMGPROC_WARN(error while processing image filter)))
3052     return -1;
}
3054
/* if there is a successor plug-in, forward the image to it */
if (strcmp(imgproc_data.successor, "nada") != 0)
3058 {
    /* create remote reference for the successor */
3060     ret = imgproc_createremoteref( &remoteobj, "1002",
                                     imgproc_data.successor);
3062     if (ret != 0)
    {
3064         IMGPROC_PRINT((
            IMGPROC_WARN(could not create remote object references)))
3066
        /* create remote refernce of the PPM node */
3068         ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                         imgproc_data.ppmnode);
3070         if (ret != 0)
        {
3072             IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))
3074             return -1;
        }
3076
        /* call the PPM for pipeline restoration */
3078         ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                         imgproc_data.successor);
3080         if (ret != 0)
        {
3082             IMGPROC_PRINT((
                IMGPROC_WARN(could not repair pipe)))
3084             rmix_remoteref_destroy( &remoteobj_ppm);
            return -1;
3086         }
3088         rmix_remoteref_destroy( &remoteobj_ppm);
        ret = imgproc_createremoteref( &remoteobj, "1002",
3090                                     imgproc_data.successor);
3092         if (ret != 0)
        {
            IMGPROC_PRINT((
```

A. Appendix

```
3094         IMGPROC_WARN(could not create remote object
3095                       references))
3096     return -1;
3097 }
3098 }
3099
3100 /* forward the processed image to the next plug-in unit */
3101 ret = rmiximgprocclient_passimage_oneway( remoteobj, imgname, blob, size);
3102 if (ret != 0)
3103 {
3104     IMGPROC_PRINT((
3105         IMGPROC_WARN(could not call remote object for passing image)))
3106
3107     /* create remote reference of the PPM node */
3108     ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
3109                                   imgproc_data.ppmnode);
3110     if (ret != 0)
3111     {
3112         IMGPROC_PRINT((
3113             IMGPROC_WARN(could not create remote object references)))
3114         return -1;
3115     }
3116
3117     /* call the PPM for pipeline restoration */
3118     ret = rmixppmclient_repairpipe( remoteobj_ppm,
3119                                     imgproc_data.successor);
3120     if (ret != 0)
3121     {
3122         IMGPROC_PRINT((
3123             IMGPROC_WARN(could not repair pipe)))
3124         rmix_remoteref_destroy( &remoteobj_ppm);
3125         return -1;
3126     }
3127
3128     rmix_remoteref_destroy( &remoteobj_ppm);
3129     ret = imgproc_createremoteref( &remoteobj, "1002",
3130                                   imgproc_data.successor);
3131     if (ret != 0)
3132     {
3133         IMGPROC_PRINT((
3134             IMGPROC_WARN(could not create remote object
3135                           references)))
3136         return -1;
3137     }
3138
3139     /* resend the image */
3140     ret = rmiximgprocclient_passimage_oneway ( remoteobj, imgname, blob,
3141                                                 size);
3142     if (ret != 0)
3143     {
3144         IMGPROC_PRINT((
3145             IMGPROC_WARN(could not call remote object for passing
3146                           image)))
3147         return -1;
3148     }
3149 }
3150 }
3151
3152 ret = rmix_remoteref_destroy( &remoteobj);
3153 if (ret != 0)
3154 {
3155     IMGPROC_PRINT((
3156         IMGPROC_WARN(could not destroy remote object references)))
```

A. Appendix

```
3158     }
3160     /* Lock image processing plug-in mutex. */
3161     if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
3162         errno = error;
3163         IMGPROC_PRINT((
3164             IMGPROC_WARN(unable to lock image processing plug-in mutex))
3165             harness_syserr());
3166         return -1;
3167     }
3168     /* store the image in the backup list if it is not stored yet */
3169     ret = imgproc_worklist_find_elementposition( imgname, worklist);
3170     if (ret == -1)
3171         imgproc_worklist_insert( imgname, size, blob, &worklist);
3172
3173     /* Unlock image processing plug-in mutex. */
3174     if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
3175     {
3176         errno = error;
3177         IMGPROC_PRINT((
3178             IMGPROC_WARN(unable to unlock image processing plug-in mutex))
3179             harness_syserr());
3180         return -1;
3181     }
3182
3183     IMGPROC_PRINT((
3184         IMGPROC_INFO(image %s successfully sent to %s),imgname
3185         ,imgproc_data.successor))
3186
3187     /* testing the reload of a plug-in */
3188     /*
3189     if (testswitch == 0)
3190     {
3191         imgproc_createremoteref( &remoteobj, "1003",imgproc_data.ppmnode);
3192         rmixppmclient_repairpipe ( remoteobj,imgproc_data.successor);
3193         rmix_remoteref_destroy( &remoteobj);
3194         testswitch = 1;
3195     }
3196     */
3197
3198 }
3199
3200 /* it there is no successor, store the processed image */
3201 else
3202 {
3203     /* construct the path */
3204     storename = (char*)malloc( sizeof(char) *
3205         (strlen(imgproc_data.targetdir) +2));
3206     if (storename == NULL)
3207     {
3208         IMGPROC_PRINT((
3209             IMGPROC_WARN(allocating of memory failed))
3210             return -1;
3211         }
3212     strcpy(storename, imgproc_data.targetdir);
3213     strcat(storename, "/");
3214
3215     tmpfile = storename;
3216
3217     /* add the filename to the path */
3218     storename = (char*)malloc( sizeof(char) * (strlen(tmpfile) +
```

A. Appendix

```
3220         strlen(imgname) +1 ));
3221     if (storename == NULL)
3222     {
3223         IMGPROC_PRINT((
3224             IMGPROC_WARN(allocating of memory failed)))
3225         return -1;
3226     }
3227     strcpy( storename, tmpfile);
3228     strcat( storename, imgname);
3229
3230     ret = imgproc_write_image( blob, size, storename);
3231     if (ret != 0)
3232     {
3233         IMGPROC_PRINT((
3234             IMGPROC_WARN(could not store image file %s), storename))
3235         FREE(tmpfile);
3236         FREE(storename);
3237         return -1;
3238     }
3239
3240     FREE(tmpfile);
3241     FREE(storename);
3242
3243     /* if the pipeline only consists of the first plug-in, first it is no
3244        pipeline and second, the images do not have to be stored */
3245
3246     /* one more image was processed, decrease the counter */
3247     imgproc_decrease_imagecounter();
3248 }
3249 return(0);
3250 }
3251
3252
3253
3254 /*****
3255  *
3256  * next section contains functions for accessing the image backup list
3257  *
3258  *****/
3259
3260 /*
3261  * Inserts an element in the backup list.
3262  *
3263  * \param  *name      Name of the image
3264  * \param  size       Size of the image
3265  * \param  *blob      Data of the image
3266  * \param  **list_pointer Pointer to the linked list
3267  * \return 0 on success or -1 on any error
3268  */
3269 #undef __FUNC__
3270 #define __FUNC__ "imgproc_worklist_insert"
3271 int imgproc_worklist_insert( char *name,
3272                             size_t size,
3273                             unsigned char *blob,
3274                             worklist_t **list_pointer)
3275 {
3276     int i;
3277     worklist_t *new_element;
3278     worklist_t *tmp_element;
3279
3280     /* allocate memory for the new element */
3281     new_element = (worklist_t*) malloc(sizeof(worklist_t));
3282     if ( new_element == NULL )
```

A. Appendix

```
3284 {
3285     IMGPROC_PRINT((
3286         IMGPROC_WARN(allocating of memory for new list element not
3287             possible)))
3288     return -1;
3289 }
3290
3291 /* allocate memory for the name string of the new element */
3292 new_element->name = (char*)malloc(sizeof(char)*(strlen(name)+1));
3293 if ( new_element->name == NULL )
3294 {
3295     IMGPROC_PRINT((
3296         IMGPROC_WARN(allocating of memory for the name string of the new
3297             list element not possible)))
3298     FREE(new_element);
3299     return -1;
3300 }
3301
3302 /* allocate memory for the image data of the new element */
3303 new_element->blob = (unsigned char*) malloc(sizeof(unsigned char) * size);
3304 if ( new_element->blob == NULL )
3305 {
3306     IMGPROC_PRINT((
3307         IMGPROC_WARN(allocating of memory for the name string of the new
3308             list element not possible)))
3309     FREE(new_element->name);
3310     FREE(new_element);
3311     return -1;
3312 }
3313
3314 /* store the information */
3315 strcpy( new_element->name, name);
3316
3317 new_element->size = size;
3318
3319 for ( i=0; i<size; i++)
3320     new_element->blob[i] = blob[i];
3321 new_element->next = NULL;
3322
3323 /* add the new element at the end of the list */
3324 if ( *list_pointer == NULL )
3325 {
3326     *list_pointer = new_element;
3327     new_element->next = NULL;
3328 }
3329 else
3330 {
3331     tmp_element = *list_pointer;
3332
3333     while ( tmp_element->next != NULL )
3334     {
3335         tmp_element = tmp_element->next;
3336     }
3337
3338     tmp_element->next = new_element;
3339     new_element->next = NULL;
3340 }
3341
3342 return 0;
3343 }
3344
```

A. Appendix

```
3346 /*
3347  * Deletes the linked list defined by the pointer.
3348  *
3349  * \param **list_pointer Pointer to the linked list
3350  */
3351 #undef __FUNC__
3352 #define __FUNC__ "imgproc_worklist_destroylist"
3353 void imgproc_worklist_destroylist( worklist_t **list_pointer)
3354 {
3355     worklist_t *tmp_pointer;
3356
3357     while( *list_pointer != NULL )
3358     {
3359         tmp_pointer = *list_pointer;
3360         *list_pointer = (*list_pointer)->next;
3361
3362         /* delete name of the element */
3363         FREE(tmp_pointer->name);
3364         FREE(tmp_pointer->blob);
3365
3366         /* delete element */
3367         FREE(tmp_pointer);
3368     }
3369     *list_pointer = NULL;
3370 }
3371
3372 /*
3373  * Deletes the element defined by the name.
3374  *
3375  * \param *name Name of the element.
3376  * \param **list_pointer Pointer to the linked list
3377  * \return 0 on success otherwise -1
3378  */
3379 #undef __FUNC__
3380 #define __FUNC__ "imgproc_worklist_delete"
3381 int imgproc_worklist_delete( char *name,
3382                             worklist_t **list_pointer)
3383 {
3384     worklist_t *tmp_pointer;
3385     worklist_t *del_element;
3386     int i=0;
3387     int position = 0;
3388     int found = 0;
3389
3390     tmp_pointer = *list_pointer;
3391
3392     /* find the entry */
3393     while ( tmp_pointer != NULL )
3394     {
3395         if ( strcmp( tmp_pointer->name, name) == 0 )
3396         {
3397             found = 1;
3398             break;
3399         }
3400         tmp_pointer = tmp_pointer->next;
3401         position++;
3402     }
3403
3404     if (found == 1)
3405     {
3406         /* it it is the first entry in the list , change the pointer to the
3407         linked list */
```


A. Appendix

```

    if ( position == 0 )
3410     {
        tmp_pointer = *list_pointer;
3412     *list_pointer = (*list_pointer)->next;

3414     /* delete name of the element */
        FREE(tmp_pointer->name);
3416     FREE(tmp_pointer->blob);

3418     /* delete element */
        FREE(tmp_pointer);
3420     }
    else
3422     {
        tmp_pointer = *list_pointer;
3424
        for ( i=0; i<position-1; i++)
3426             tmp_pointer = tmp_pointer->next;

3428         del_element = tmp_pointer->next;

3430         tmp_pointer->next = del_element->next;

3432         FREE(del_element->name);
        FREE(del_element->blob);
3434         FREE(del_element);
    }
3436 }
    return 0;
3438 }

3440 /*
3442  * Prints the names of the list elements.
    *
3444  * \param *list_pointer Pointer to the linked list
    */
3446 #undef __FUNC__
#define __FUNC__ "imgproc_worklist_printlist"
3448 void imgproc_worklist_printlist( worklist_t *list_pointer)
    {
3450     worklist_t *current_element;

3452     current_element = list_pointer;
    while ( current_element != NULL )
3454     {
        IMGPROC_PRINT((
3456             IMGPROC_INFO(name = %s\n), current_element->name))
        current_element = current_element->next;
3458     }
    }
3460

3462 /*
    * Finds the position of an entry.
3464  *
    * \param *char          Entry to find
3466  * \param *list_pointer  Pointer to the linked list
    * \return               The position or otherwise -1
3468  */
#define __FUNC__
3470 #define __FUNC__ "imgproc_worklist_find_elementposition"
    int imgproc_worklist_find_elementposition( char *entry ,
```

A. Appendix

```
3472                                     worklist_t *list_pointer)
3473 {
3474     worklist_t *current_element;
3475     int position = 0;
3476
3477     current_element = list_pointer;
3478     while ( current_element != NULL )
3479     {
3480         if ( strcmp( current_element->name, entry) == 0 )
3481             return position;
3482         current_element = current_element->next;
3483         position++;
3484     }
3485
3486     return -1;
3487 }
3488
3489 /*****
3490 *
3491 * END OF FILE
3492 *
3493 *****/
```

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Reading, 6-March-2006

Ronald Baumann