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

List of Tables

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.

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.

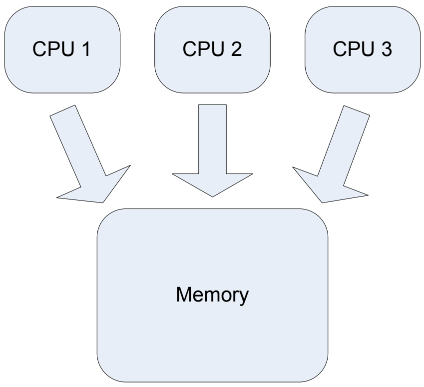

Figure 2.1.: Shared Memory System

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.

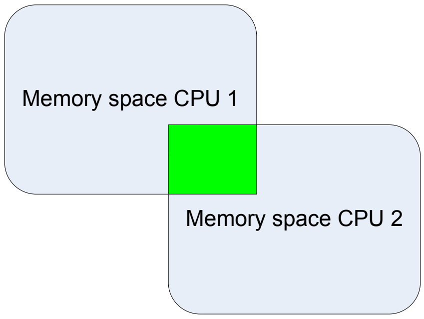

Figure 2.2.: Overlapping Memory Space

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.

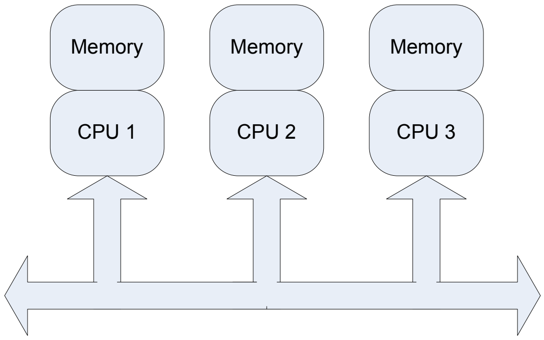

Figure 2.3.: Distributed Memory System

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).

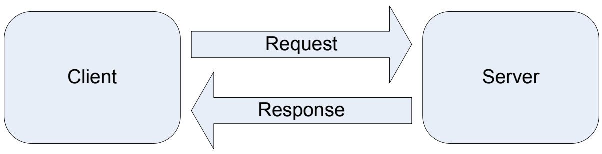

Figure 2.4.: Client/Server Model

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 exists, 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]

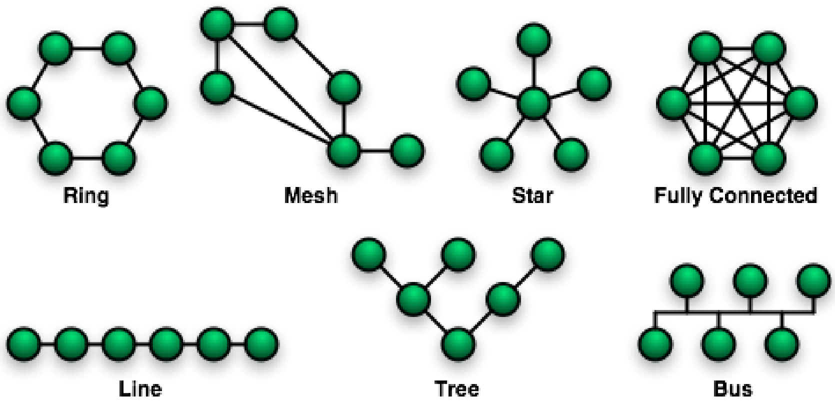

Figure 2.5.: Network Topologies [Wik06c]

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 encapsulation. [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

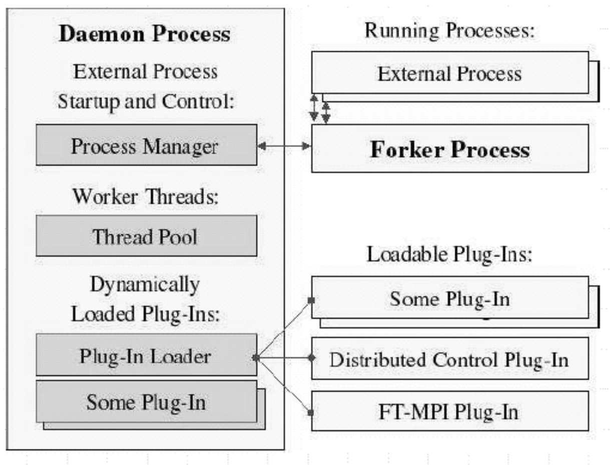

Figure 2.6.: Lightweight Kernel Design [EG05a]

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 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]

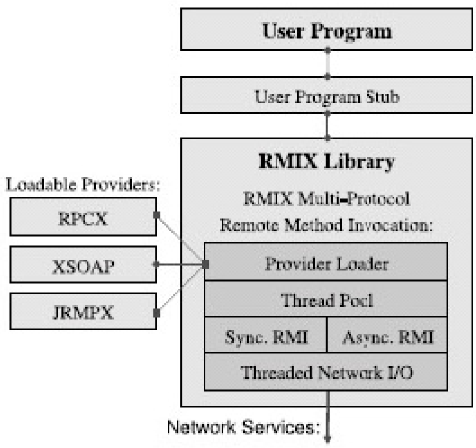

Figure 2.7.: RMIX Framework Architecture [EG05b]

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]

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]

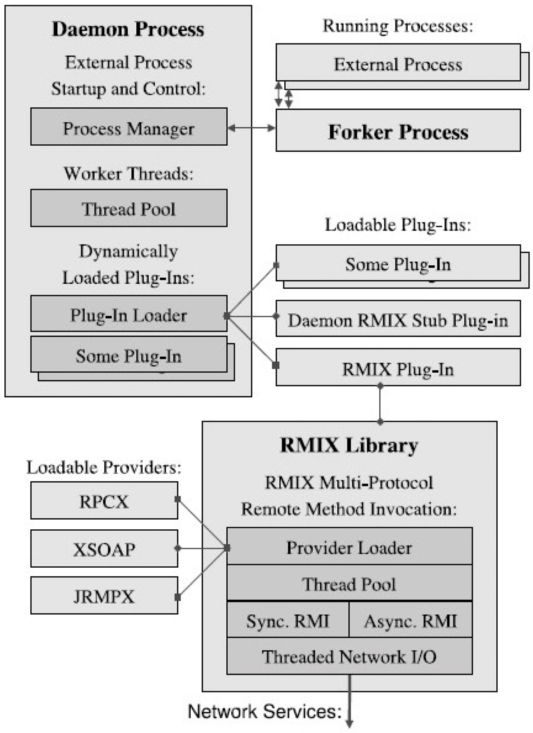

Figure 2.8.: RMIX Plug-in for Harness [EG05b]

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 software 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]

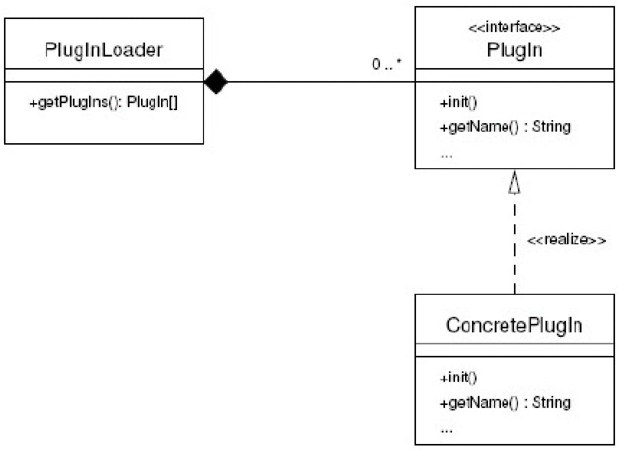

Figure 2.9.: UML Class Diagram for a Plug-in Pattern [MMS02]

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 platforms 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.

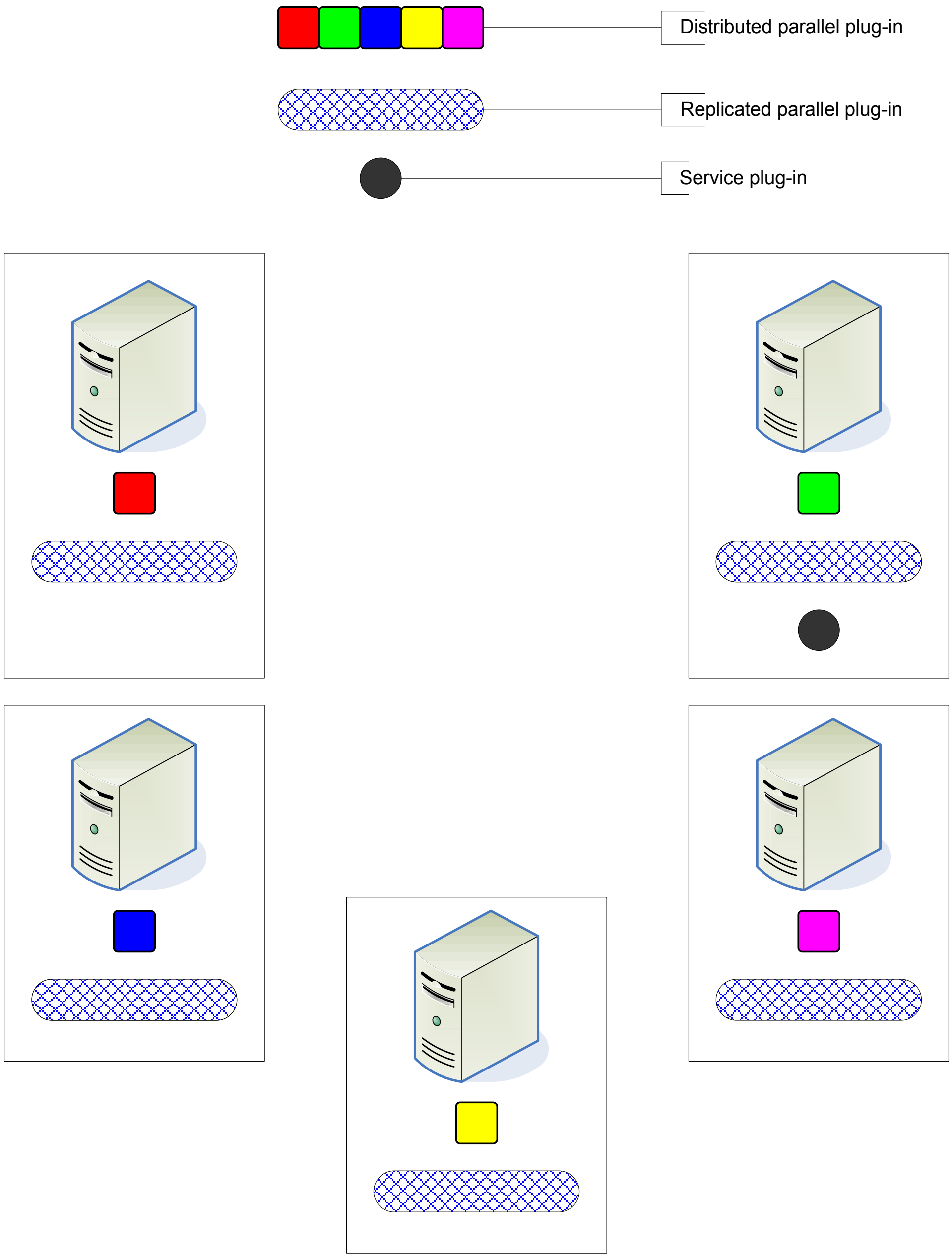

Figure 2.10.: Types of Parallel Plug-ins

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 compositions 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

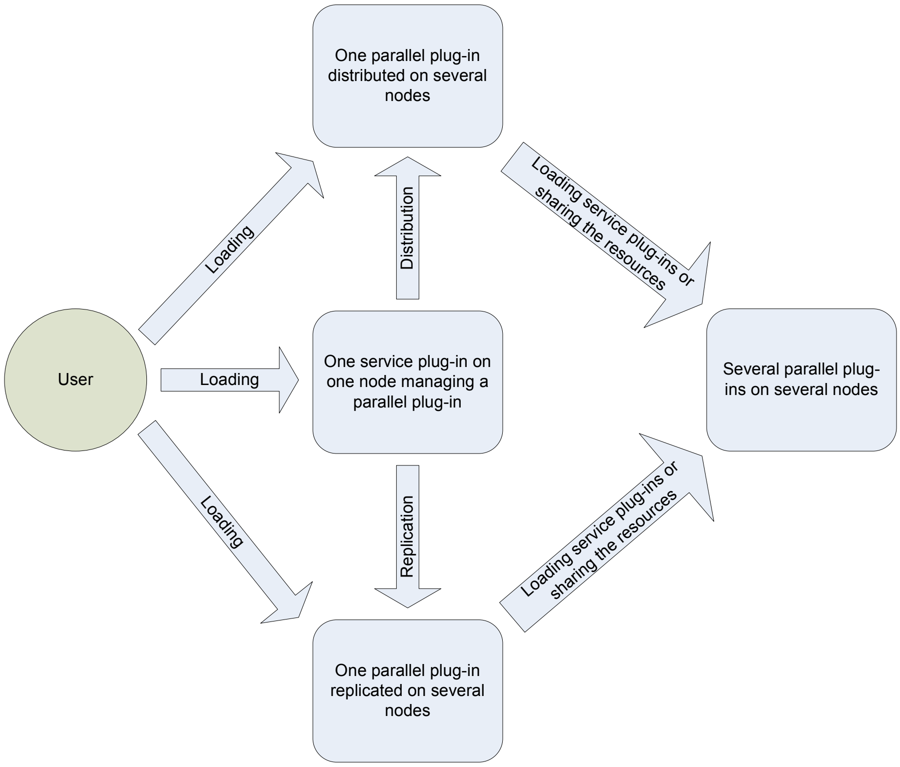

Figure 2.11.: Loading of Parallel Plug-ins

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

$$\int_{t_0}^{t_1} f(x)\mathrm{d}x = \lim_{n\to\infty} \frac{t_1 - t_0}{n} \sum_{i=1}^{n} f(x_i) \tag{2.1}$$

where x_i are supporting points.

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

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

$$I = \int \mathrm{d}x_1 \int \mathrm{d}x_2 \cdots \int \mathrm{d}x_D \quad f(x_1, x_2, \cdots, x_D) \tag{2.3}$$

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

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

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

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

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.

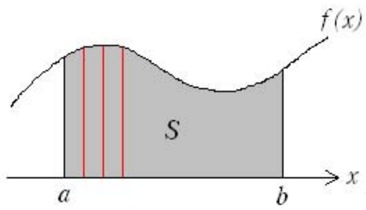

Figure 2.12.: Integral Divided into Chunks

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.

$$I = \int_a^b g(x)\mathrm{d}x \tag{2.6}$$

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]

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

$$E\left[g(x)\right] = \int_{-\infty}^{+\infty} f(x) \cdot g(x) \mathrm{d}x = \frac{1}{b-a} \int_{a}^{b} 1 \cdot g(x) \mathrm{d}x \tag{2.8}$$

$$\int_{a}^{b} g(x) \mathrm{d}x = (b-a) E\left[g(x)\right] \tag{2.9}$$

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

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

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

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

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.

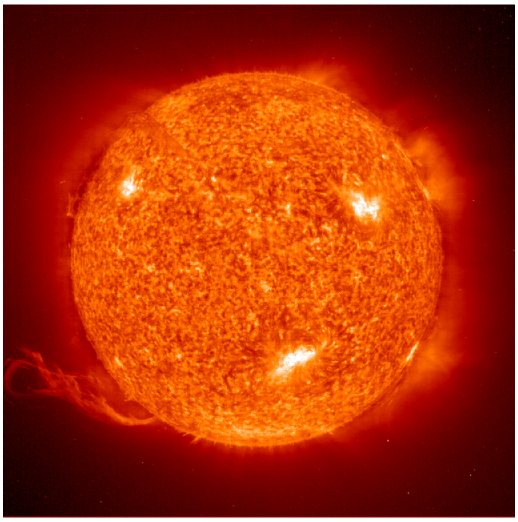

Figure 2.13.: Source Image

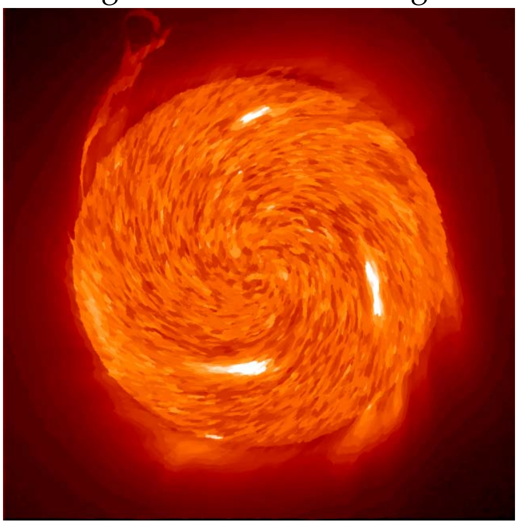

Figure 2.14.: Processed Image

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.

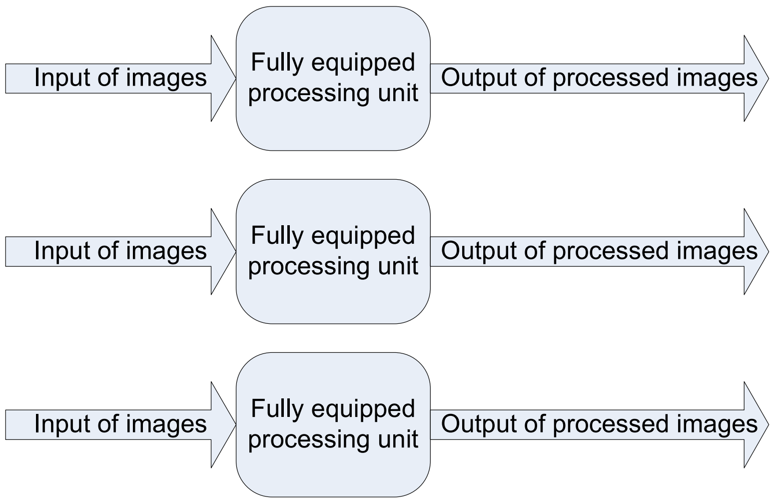

Figure 2.15.: Image Processing with Copied, Fully-equipped Processing Units

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]

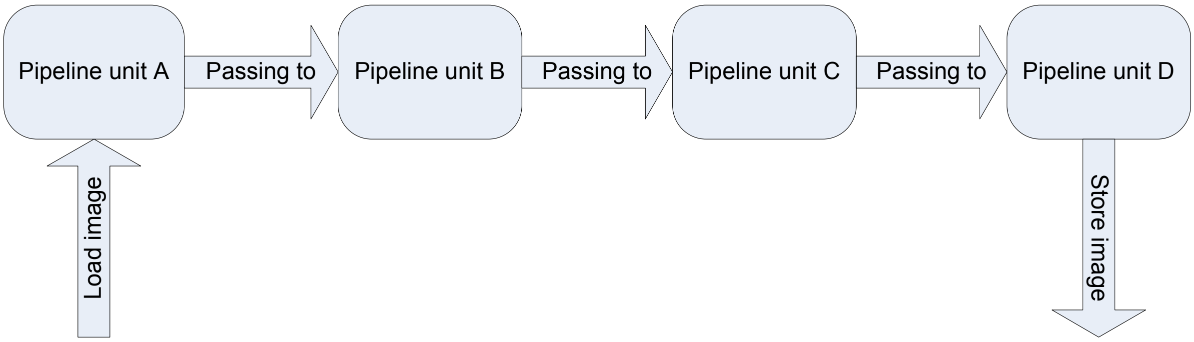

Figure 2.16.: Image Processing Pipeline

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.

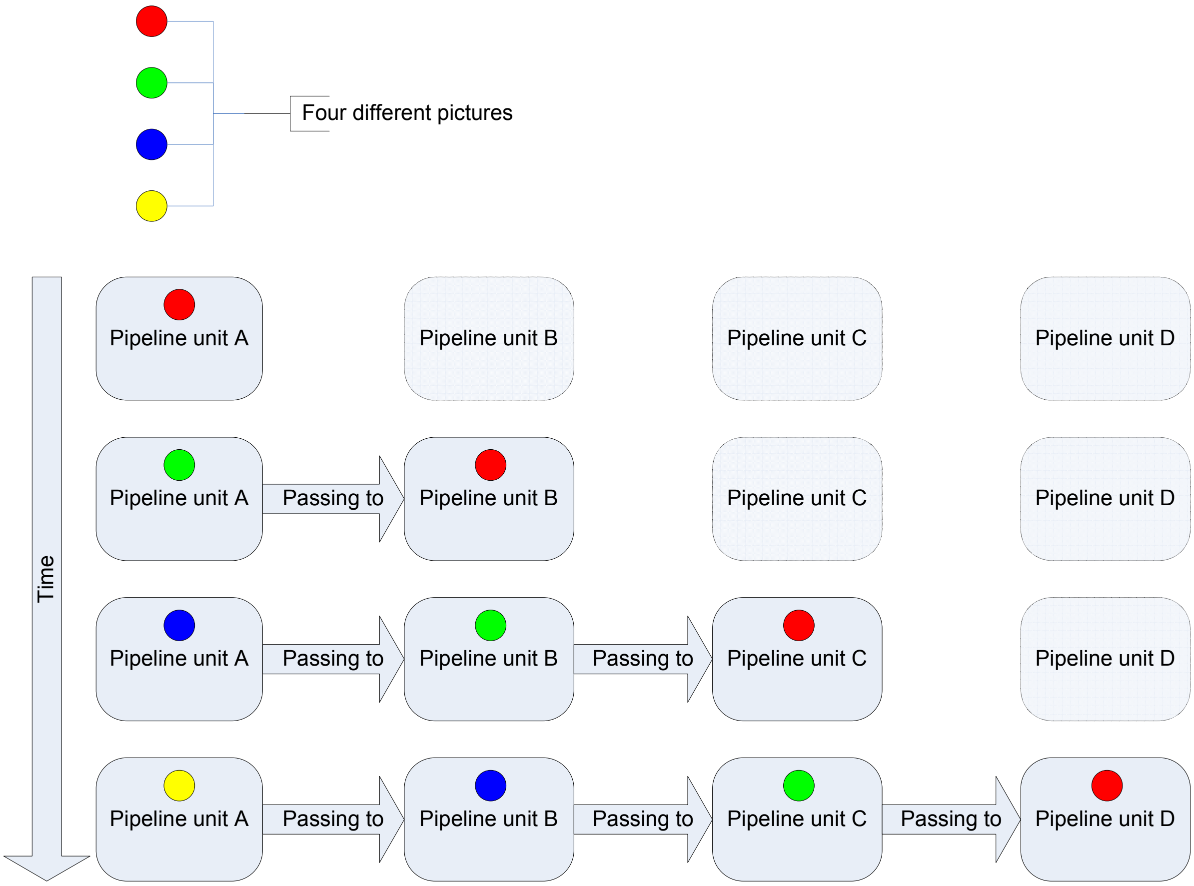

Figure 2.17.: Filling the Image Processing Pipeline with Data

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

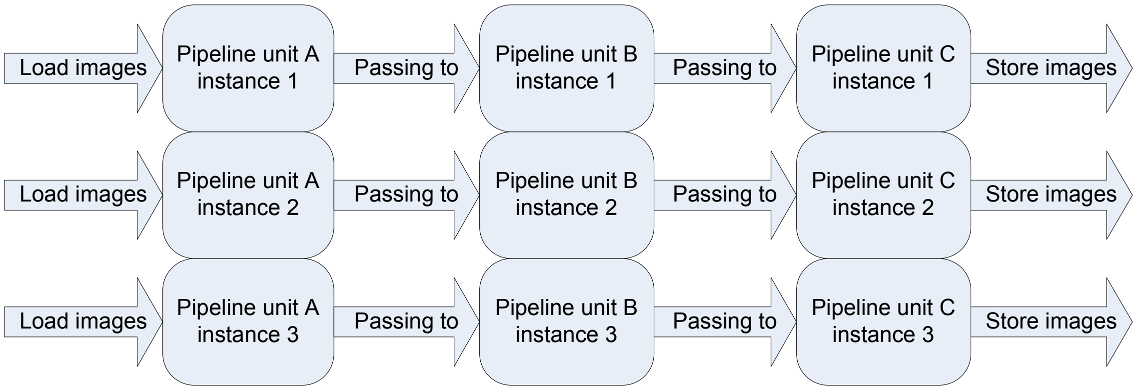

Figure 2.18.: Multiple Image Processing Pipeline

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.

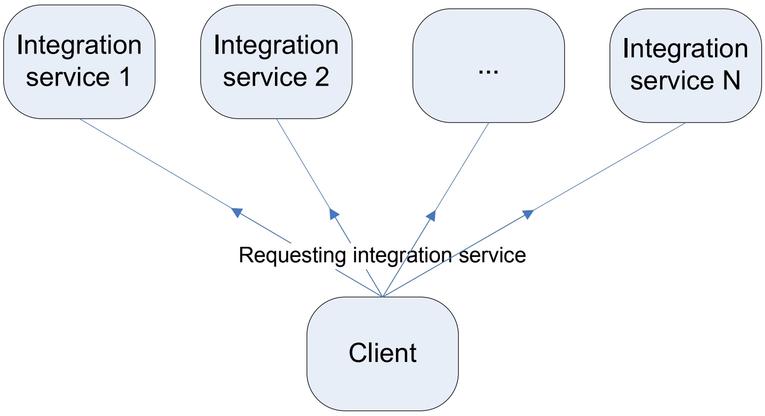

Figure 2.19.: Concept of the Parallel Plug-in for Integration

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.

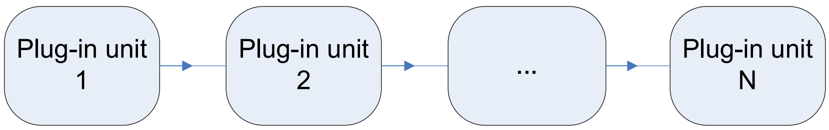

Figure 2.20.: Concept of a Parallel Plug-in for Image Processing

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.

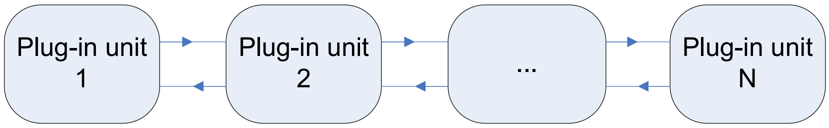

Figure 2.21.: Image Processing Pipeline with Acknowledgment Functionality

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.

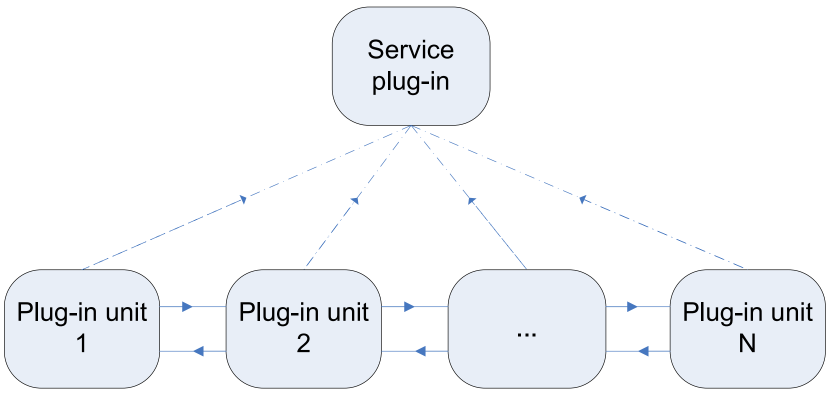

Figure 2.22.: Image Processing Pipeline with Service Plug-in

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

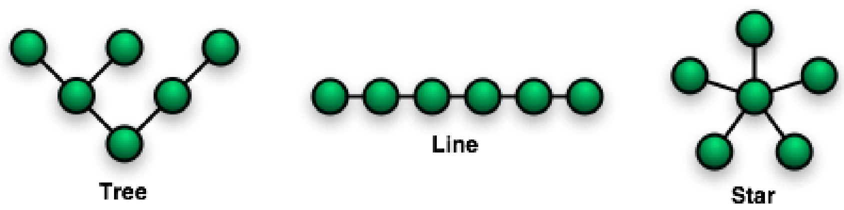

Figure 2.23.: Possible Ways of Loading Parallel Plug-ins

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 subtree 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 application 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.

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 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.

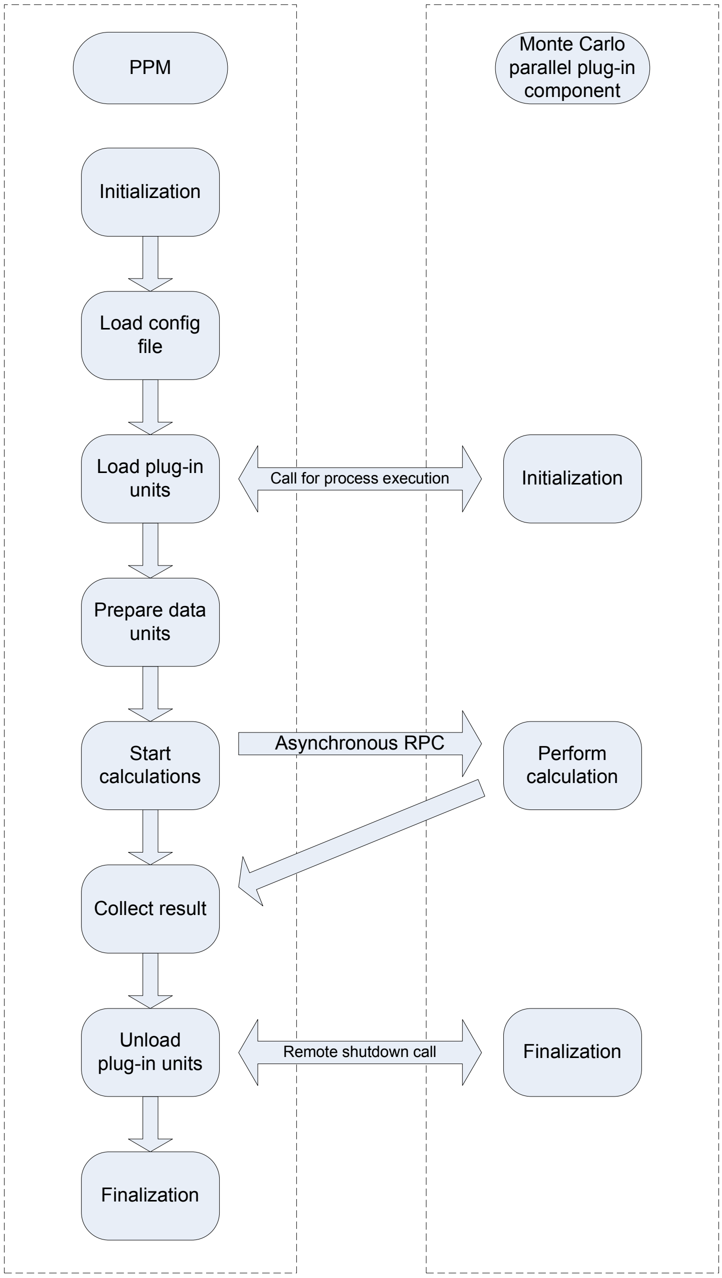

Figure 3.1.: Life-cycle of the Monte Carlo Integration Application

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.

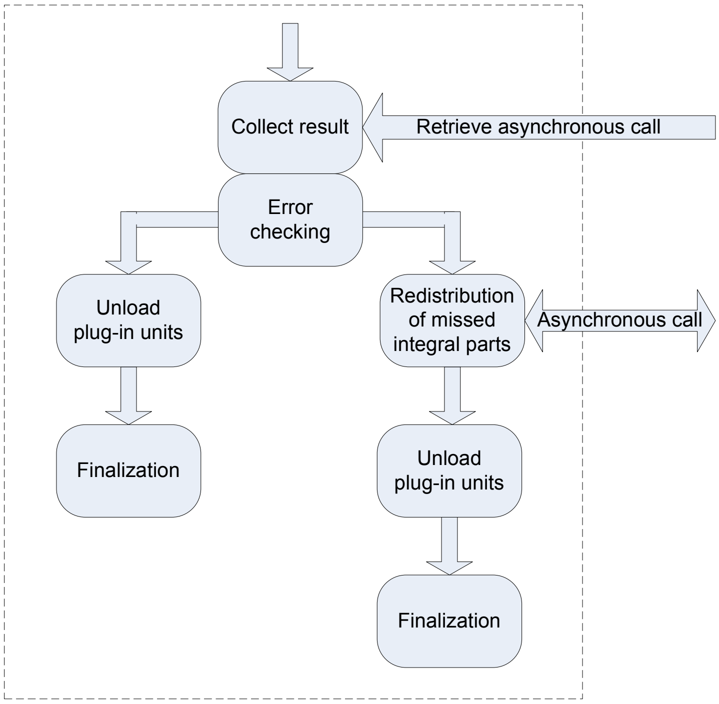

Figure 3.2.: Monte Carlo Integration with Additional Failure Handling

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.

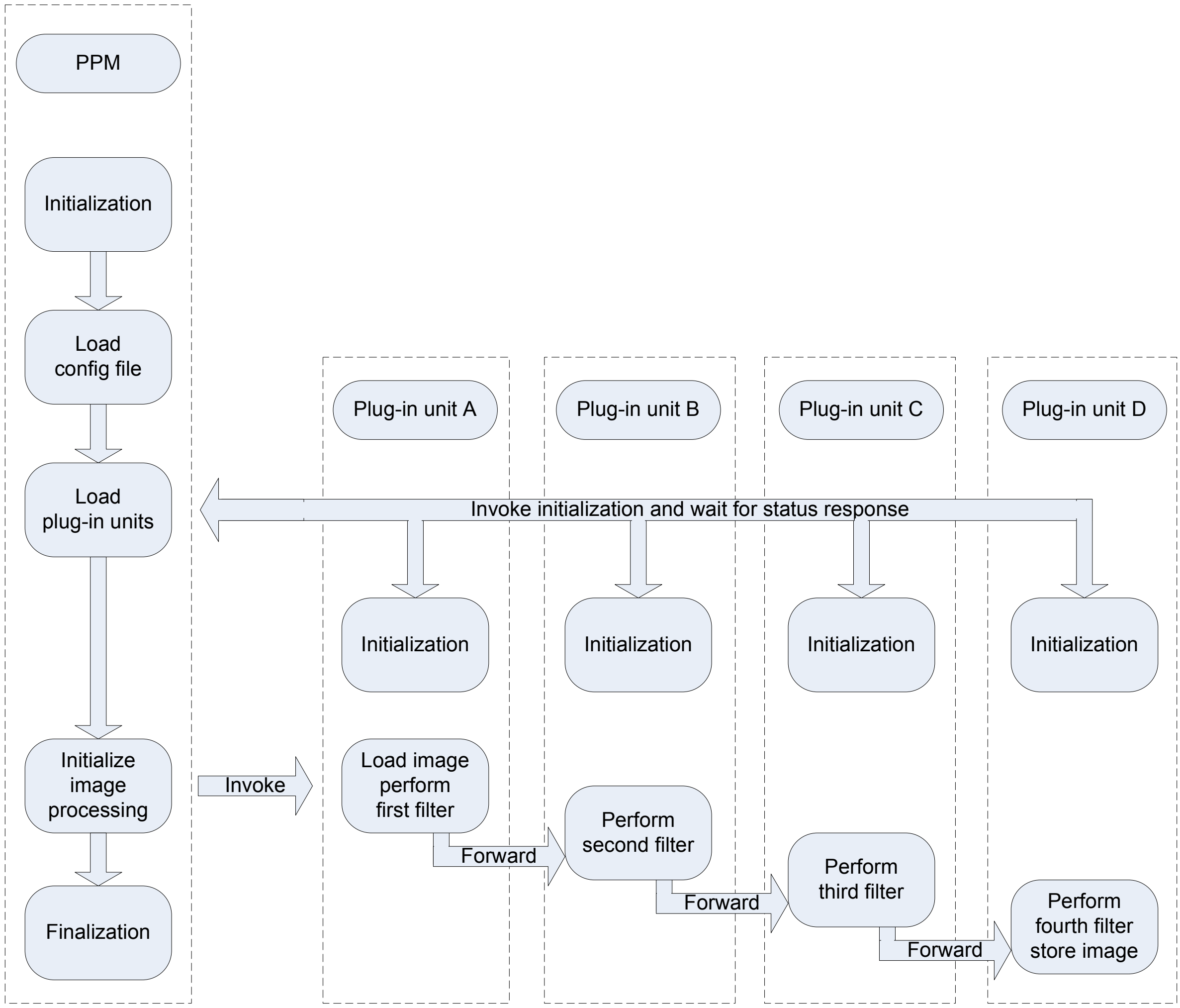

Figure 3.3.: Life-cycle of the Image Processing Pipeline Application

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 application 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 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.

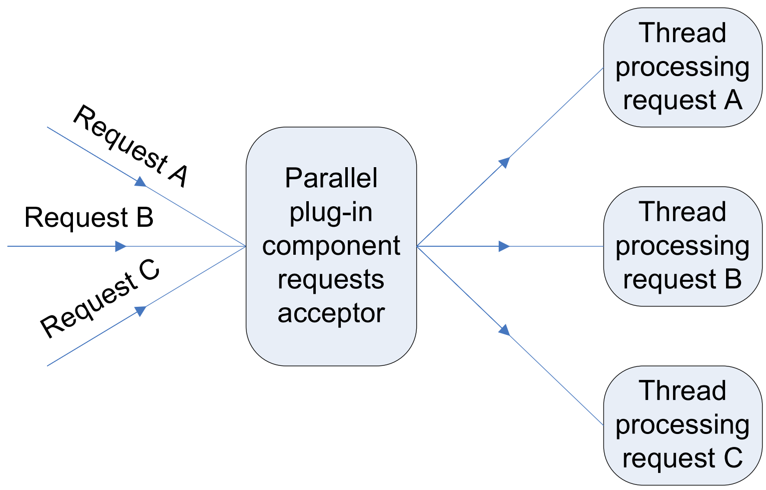

Figure 3.4.: The Harness Thread Pool and Parallel Plug-ins

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

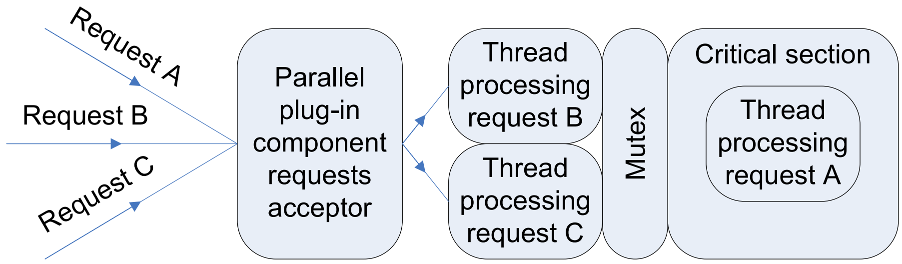

Figure 3.5.: Thread Approach with Critical Section

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.

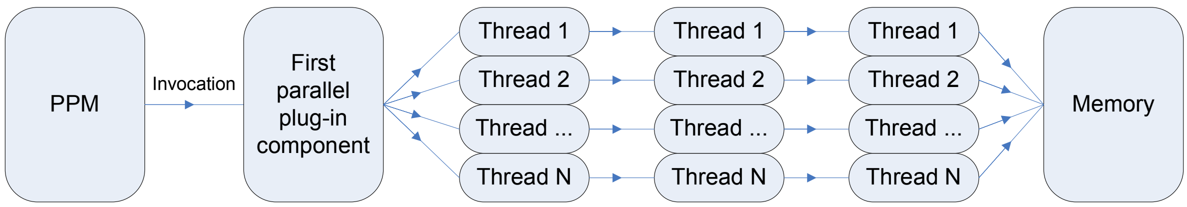

Figure 3.6.: Image Processing Pipeline Implemented with Threads

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 possible 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

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

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

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.

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.

$$I = \int_{-2}^{2} \left(x^2 - 4\right) \mathrm{d}x \tag{3.1}$$

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

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

$$I = -\frac{32}{3} = -10.667 \tag{3.4}$$

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 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.

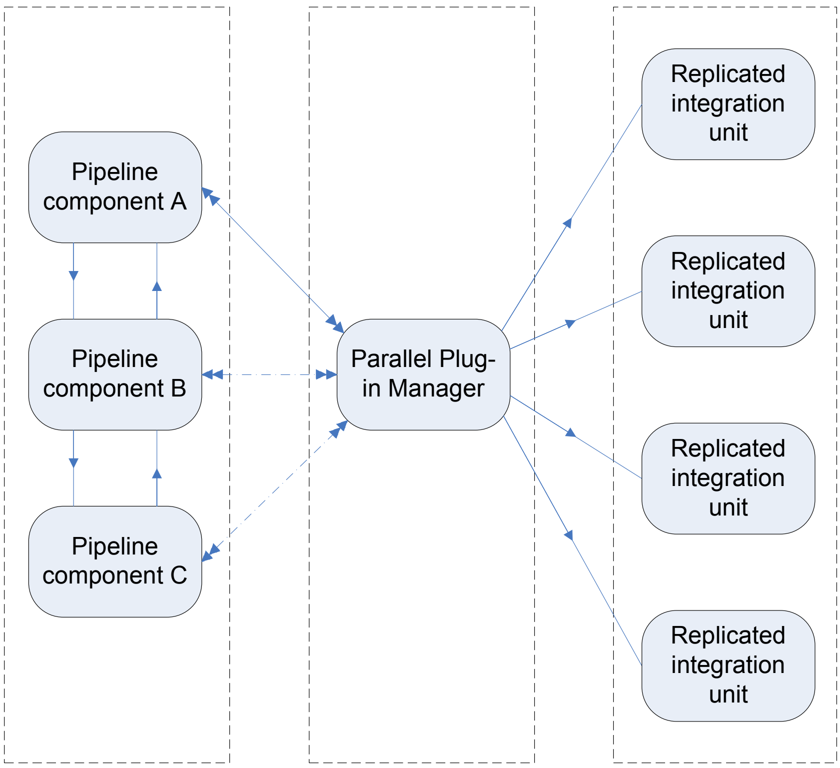

Figure 4.1.: Architecture of the Parallel Plug-in Prototype Suite

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-

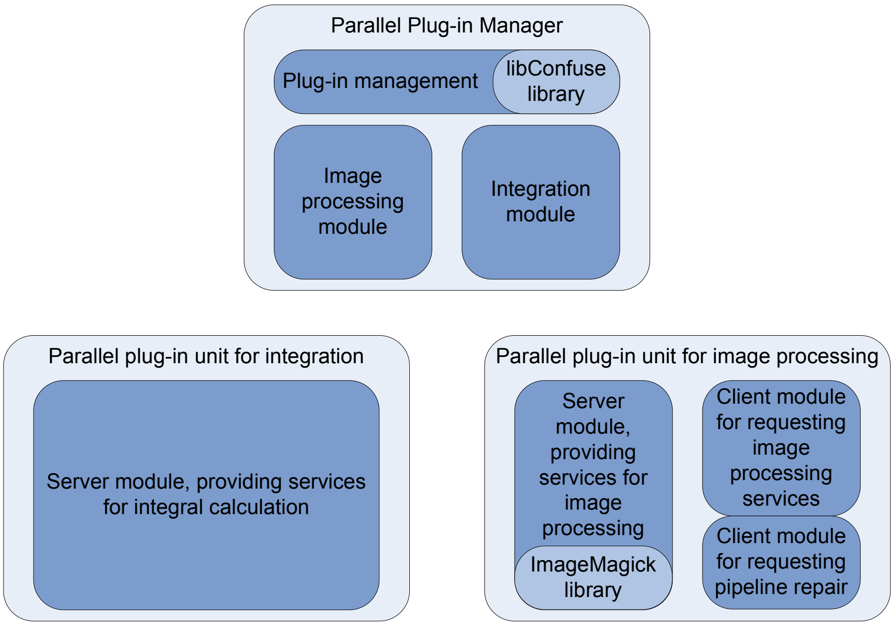

Figure 4.2.: Design of the Prototype Suite Components

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, its 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

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.

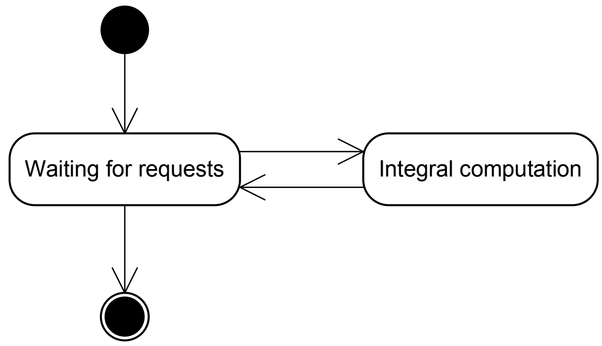

Figure 4.3.: Finite State Diagram for Integration Plug-in Component

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).

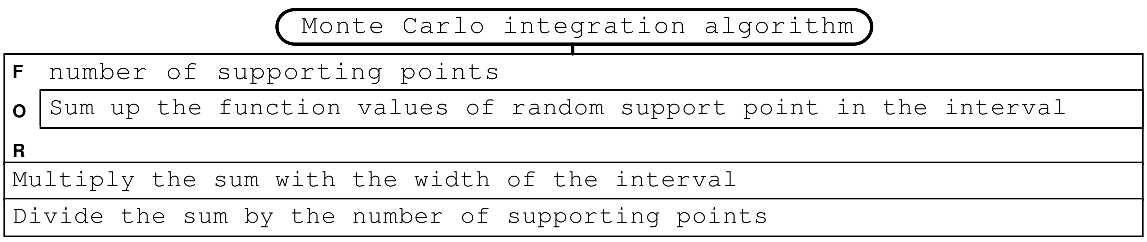

Figure 4.4.: Nassi-Schneidermann Diagram for the Integration Algorithm

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.

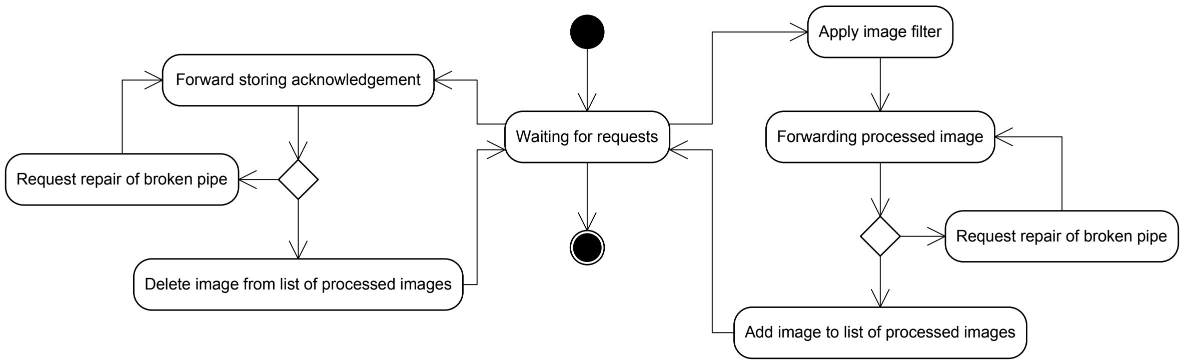

Figure 4.5.: Finite state diagram pipeline component

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.

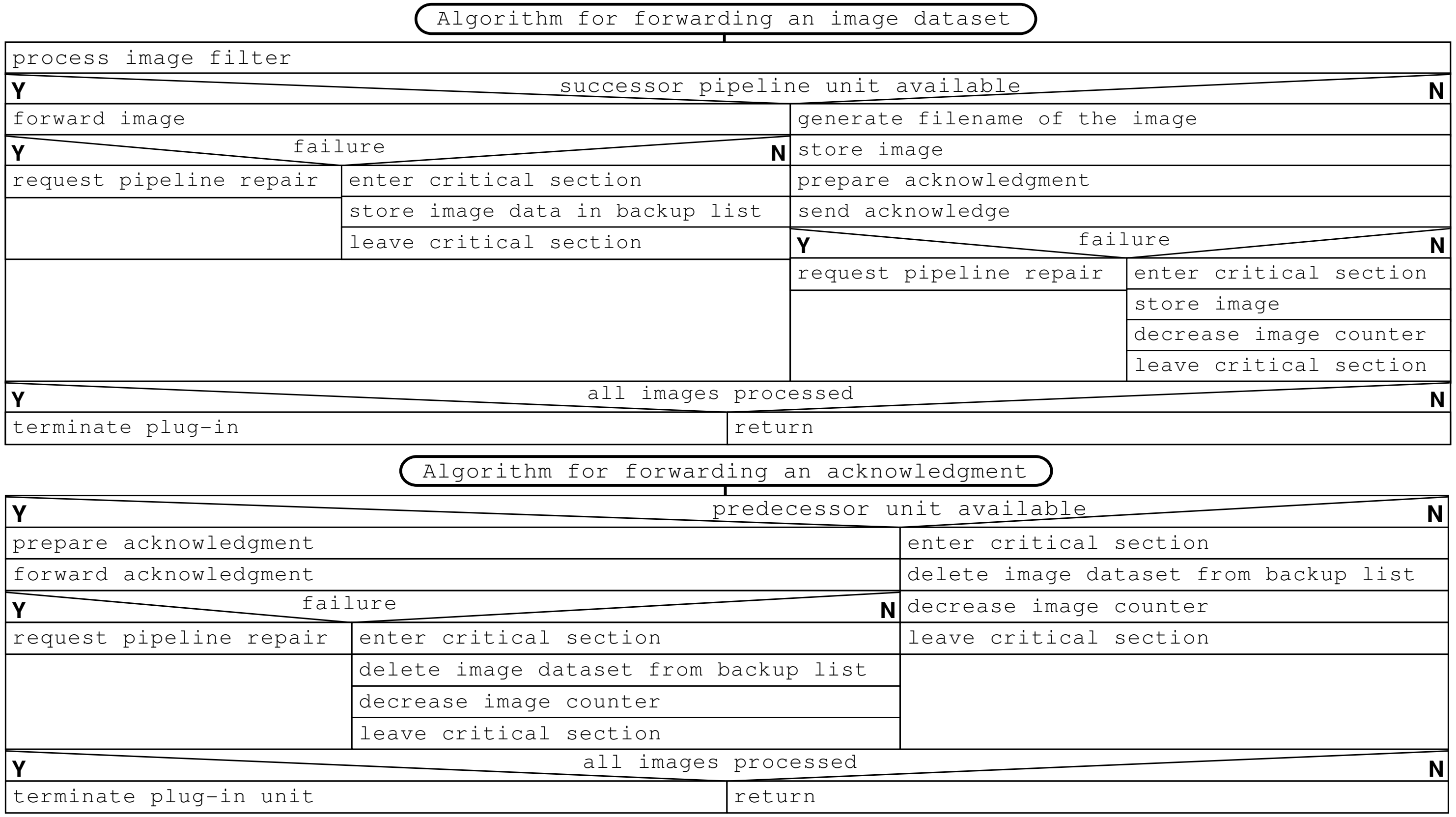

Figure 4.6.: Nassi-Schneidermann Diagrams for Image and Acknowledgment Passing Algorithms

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

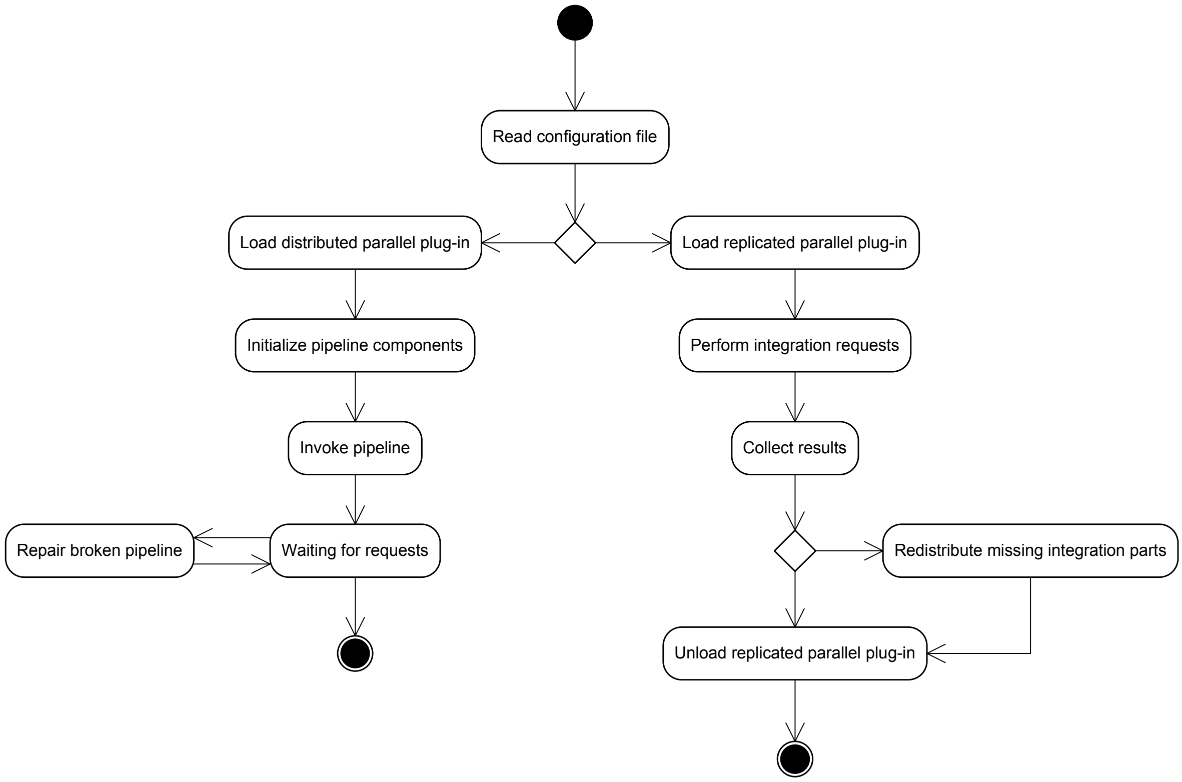

Figure 4.7.: Finite State Diagram of the Parallel Plug-in Manager

The program flow of the parallel plug-in manger 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 instruction are carried out. On page 88 figure 4.8 gives an over view 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.

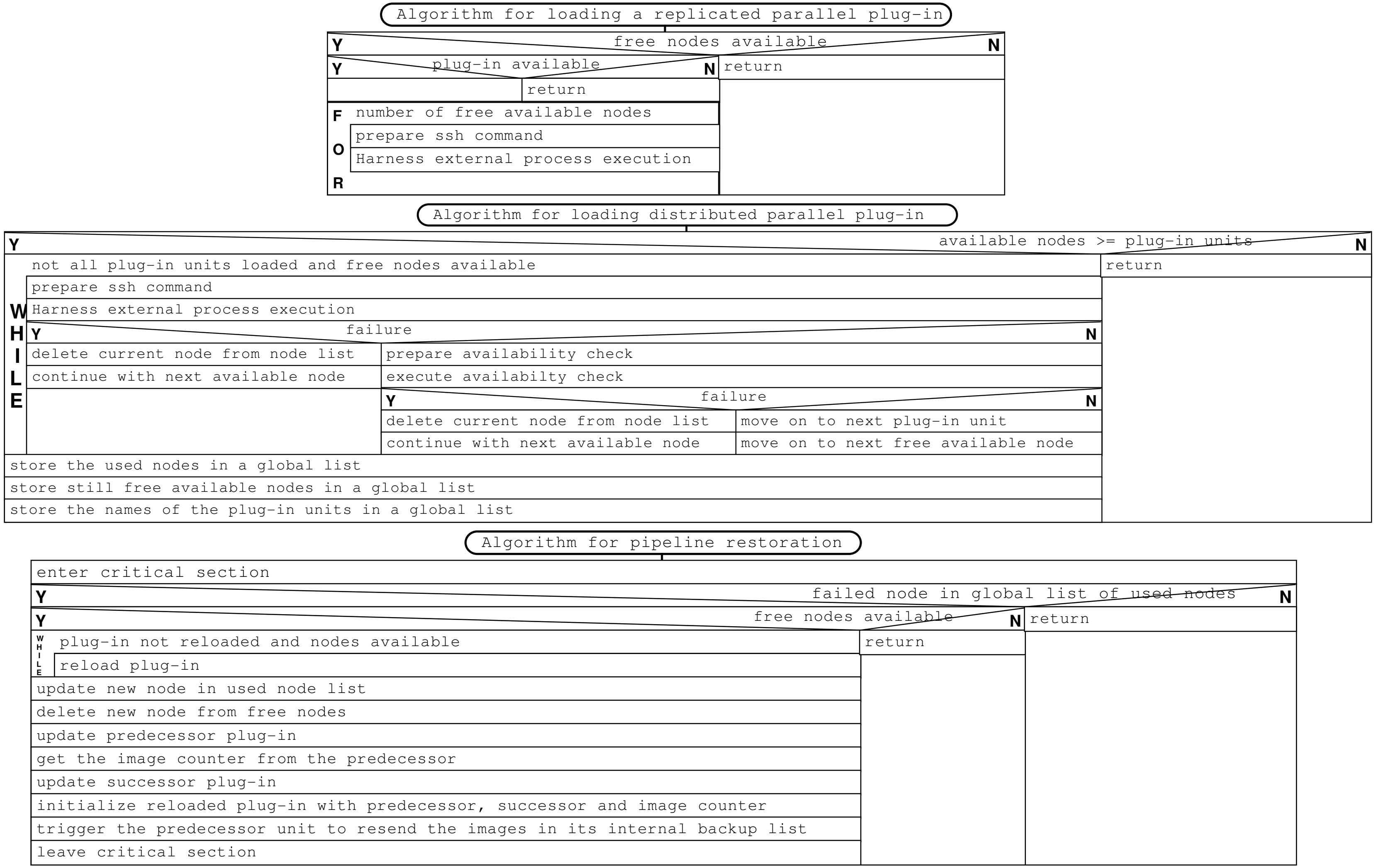

Figure 4.8.: Nassi-Schneidermann Diagrams for Parallel Plug-in Loading Algorithms and Pipeline Restoration

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.

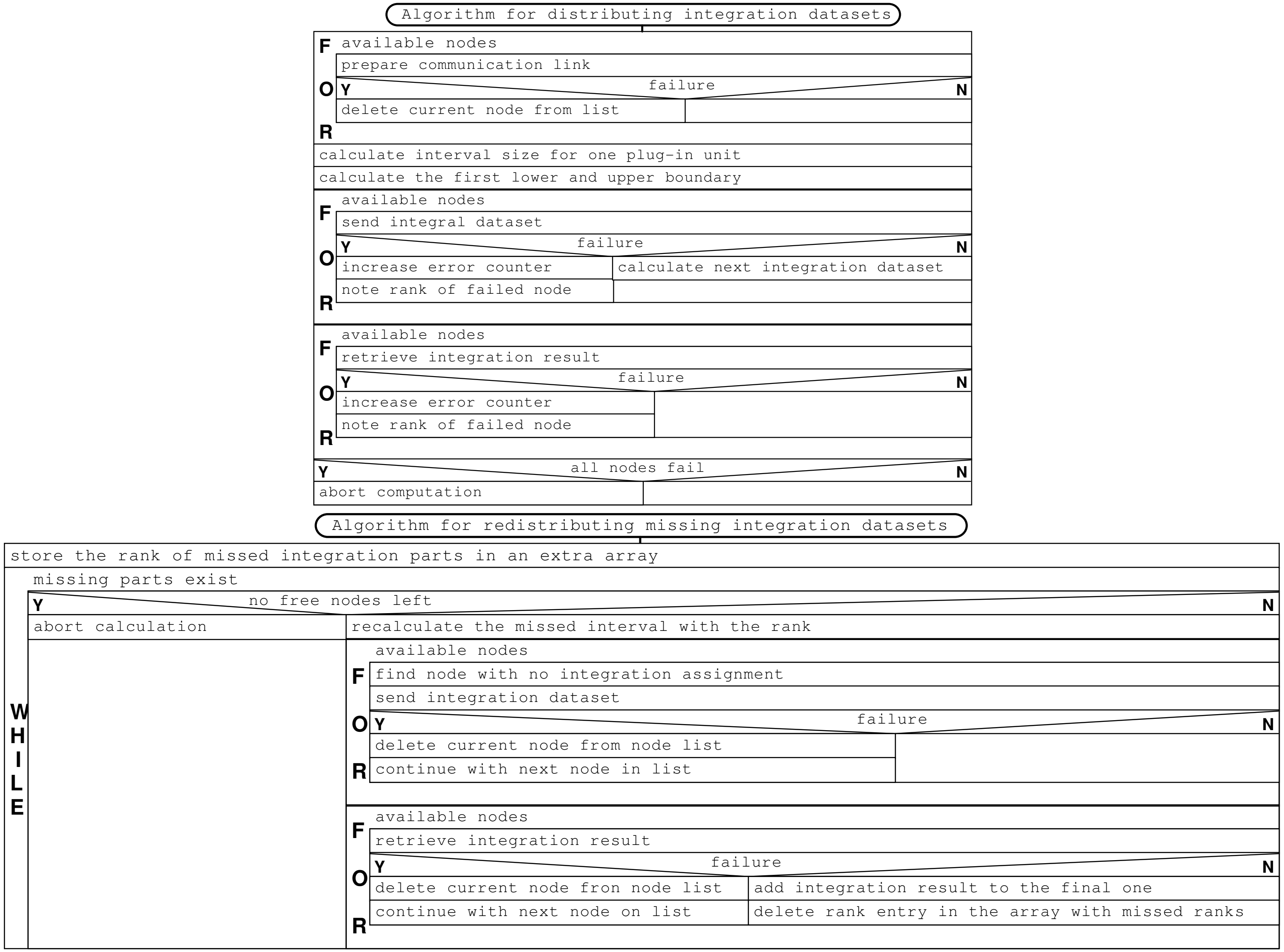

Figure 4.9.: Nassi-Schneidermann Diagram for Integration Distribution and Redistribution Algorithms

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, 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 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.

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 Papadopoulous, Stephen L. Scott, and Vaidy Sunderam. *HARNESS: 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 Metacomputing 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 Performace Computing Workshop (HAPCW) 2005*, Santa Fe, NM, USA, October 2005.

[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. *HARNESS: Reconfigurable Distributed Environments for Heterogenous 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].

[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. Milojicic, 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 Plug-in-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).

[Sch03] Prof. Wolfgang Schebesta. Data and compute networks, April 2003. lecture notes.

[SL] Computer Science and Mathematics Devision 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 Devision 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 installation 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 subdirectories 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

```
# this is the configuration file for the PPM

nodelist
{
    adresses = {  "h1", "martha", "h2", "h3" }
}

replicated
{
    plugins = {libintegral.0.0.0.so}
}

distributed
{
    plugins = { "libimgproc.0.0.0.so",
                "libimgproc.0.0.0.so",
                "libimgproc.0.0.0.so"
                }
}
```

```
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 integration 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.

$$\int_{a}^{b} f(x)\mathrm{d}x \tag{A.1}$$

$$\int_{-2}^{2} \left(0x^9 + 0x^8 + 0x^7 + 0x^6 + 0x^5 + 0x^4 + 0x^3 + 1x^2 + 0x^1 - 4x^0\right)\mathrm{d}x \tag{A.2}$$

$$\int_{-2}^{2} \left(x^2 - 4\right)\mathrm{d}x \tag{A.3}$$

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

```
# this is the configuration file for the image processing pipeline

sourcedir = "/home/ronald/parallelplugins/images"
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 "parallelplugins/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 dotes 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

```
ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
info:libppm:1031:ppm.c:231:ppm_init:libppm is starting
info:libppm:1031:ppm.c:436:ppm_init_rmix:start initialization of RMIX
...
info:libplutils:1031:readconf.c:114:configuration_read_file:start reading configuration file
info:libplutils:1031:readconf.c:388:configlist_print_list:name = h1
info:libplutils:1031:readconf.c:388:configlist_print_list:name = h2
info:libplutils:1031:readconf.c:388:configlist_print_list:name = h3
info:libplutils:1031:readconf.c:388:configlist_print_list:name = joshua
...
info:libplutils:1031:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
...
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
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 = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:1031:readconf.c:388:configlist_print_list:
     name = /home/ronald/parallelplugins/data/image.input
...
info:libppm:1031:ppm.c:570:ppm_start:mode = 1
info:libppm:1031:ppm.c:572:ppm_start:ppmnode = kid
...
info:libppm:1031:ppm.c:716:ppm_load_harnesskernel_replicated:
     start loading harness kernel and parallel plug-in
```

```
info:libppm:1031:ppm.c:1422:ppm_read_integralinput:start reading input file
...
info:libppm:1031:ppm.c:1459:ppm_read_integralinput:input file is opened
info:libppm:1031:ppm.c:1580:ppm_schedule_integral:start scheduling
...
info:libintegral:4813:integral.c:202:integral_init:libintegral is starting
info:libintegral:4813:integral.c:416:integral_init_rmix:start initialization of RMIX
...
info:libintegral:4855:integral.c:202:integral_init:libintegral is starting
info:libintegral:4855:integral.c:416:integral_init_rmix:start initialization of RMIX
...
info:libintegral:4855:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:4813:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:3031:integral.c:202:integral_init:libintegral is starting
info:libintegral:3031:integral.c:416:integral_init_rmix:start initialization of RMIX
...
info:libintegral:4785:integral.c:202:integral_init:libintegral is starting
info:libintegral:4785:integral.c:416:integral_init_rmix:start initialization of RMIX
...
info:libintegral:4785:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:3031:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:4855:integral.c:659:rmixintegral_integration:integral is -1.667648
info:libintegral:4813:integral.c:659:rmixintegral_integration:integral is -3.665542
info:libintegral:4785:integral.c:659:rmixintegral_integration:integral is -3.669111
info:libintegral:3031:integral.c:659:rmixintegral_integration:integral is -1.663693
...
info:libppm:1031:ppm.c:2126:ppm_schedule_integral:++++++++++++++++++++++++++++++++++++
info:libppm:1031:ppm.c:2128:ppm_schedule_integral:integral = -10.665995
info:libppm:1031:ppm.c:2130:ppm_schedule_integral:++++++++++++++++++++++++++++++++++++
...
info:libintegral:4855:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:4855:integral.c:469:integral_fini_rmix:start finalization of RMIX
...
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
...
info:libintegral:4785:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:4785:integral.c:469:integral_fini_rmix:start finalization of RMIX
...
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
...
info:libintegral:4855:integral.c:499:integral_fini_rmix:
      RMIX finalized and integral plug-in unexported
...
info:libintegral:4813:integral.c:499:integral_fini_rmix:
      RMIX finalized and integral plug-in unexported
...
info:libintegral:4785:integral.c:499:integral_fini_rmix:
      RMIX finalized and integral plug-in unexported
...
info:libintegral:3031:integral.c:499:integral_fini_rmix:
      RMIX finalized and integral plug-in unexported
```

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

```
ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
info:libppm:28193:ppm.c:231:ppm_init:libppm is starting
info:libppm:28193:ppm.c:436:ppm_init_rmix:start initialization of RMIX
...
info:libppm:28193:ppm.c:471:ppm_init_rmix:RMIX initialized
info:libplutils:28193:readconf.c:114:configuration_read_file:start reading configuration file
info:libplutils:28193:readconf.c:388:configlist_print_list:name = h1
info:libplutils:28193:readconf.c:388:configlist_print_list:name = martha
info:libplutils:28193:readconf.c:388:configlist_print_list:name = h2
info:libplutils:28193:readconf.c:388:configlist_print_list:name = h3
...
info:libplutils:28193:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
...
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
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 = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:28193:readconf.c:388:configlist_print_list:
      name = /home/ronald/parallelplugins/data/image.input
...
info:libppm:28193:ppm.c:570:ppm_start:mode = 1
info:libppm:28193:ppm.c:572:ppm_start:ppmnode = kid
info:libppm:28193:ppm.c:716:ppm_load_harnesskernel_replicated:
      start loading harness kernel and parallel plug-in
info:libppm:28193:ppm.c:1422:ppm_read_integralinput:start reading input file
...
info:libppm:28193:ppm.c:1459:ppm_read_integralinput:input file is opened
ssh: martha: Name or service not known
...
info:libppm:28193:ppm.c:1580:ppm_schedule_integral:start scheduling
...
info:libintegral:6388:integral.c:202:integral_init:libintegral is starting
...
info:libintegral:6388:integral.c:416:integral_init_rmix:start initialization of RMIX
info:libintegral:6534:integral.c:202:integral_init:libintegral is starting
...
info:libintegral:6534:integral.c:416:integral_init_rmix:start initialization of RMIX
...
info:libintegral:6534:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:6388:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:4918:integral.c:202:integral_init:libintegral is starting
...
info:libintegral:4918:integral.c:416:integral_init_rmix:start initialization of RMIX
...
info:libintegral:4918:integral.c:451:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
warn:librmix:28193:tcpip4.c:264:rmix_tcpip4_resolve:
      unable to lookup local name 'martha'
warn:librmix-rpcx:28193:rmix-rpcx.c:2655:rmix_rpcx_remoteref_create:
      unable to resolve address
warn:librmix:28193:remoteref.c:975:rmix_remoteref_create5:
      unable to create remote object reference
info:librmix-rpcx:28193:rmix-rpcx.c:328:rmix_rpcx_fini:rmix-rpcx shutdown
```

```
warn:librmix:28193:remoteref.c:1266:rmix_remoteref_create6:
      unable to create remote object reference
warn:libppm:28193:ppm.c:1671:ppm_schedule_integral:
      could not create remote object references
...
info:libintegral:6388:integral.c:659:rmixintegral_integration:integral is -2.773682
info:libintegral:6534:integral.c:659:rmixintegral_integration:integral is -5.132638
info:libintegral:4918:integral.c:659:rmixintegral_integration:integral is -2.756107
...
info:libppm:28193:ppm.c:2126:ppm_schedule_integral:++++++++++++++++++++++++++++++++++++
info:libppm:28193:ppm.c:2128:ppm_schedule_integral:integral = -10.662427
info:libppm:28193:ppm.c:2130:ppm_schedule_integral:++++++++++++++++++++++++++++++++++++
...
warn:libppm:28193:ppm.c:1236:ppm_createremoteref:
      could not create remote object references
warn:libppm:28193:ppm.c:1142:ppm_unload_harnesskernel_soft:
      could not create remote object references for martha
...
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
...
info:libintegral:6534:integral.c:321:integral_fini:libintegral is shutting down
info:libintegral:6534:integral.c:469:integral_fini_rmix:start finalization of RMIX
...
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
...
info:libintegral:6388:integral.c:499:integral_fini_rmix:
      RMIX finalized and integral plug-in unexported
...
info:libintegral:6534:integral.c:499:integral_fini_rmix:
      RMIX finalized and integral plug-in unexported
...
info:libintegral:4918:integral.c:499:integral_fini_rmix:
      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
...
info:libppm:4936:ppm.c:200:ppm_init:libppm is starting
info:libppm:4936:ppm.c:391:ppm_init_rmix:start initialization of RMIX
...
info:libppm:4936:ppm.c:404:ppm_init_rmix:RMIX initialized
info:libplutils:4936:readconf.c:110:configuration_read_file:start reading configuration file
info:libplutils:4936:readconf.c:372:configlist_print_list:name = h1
info:libplutils:4936:readconf.c:372:configlist_print_list:name = h2
info:libplutils:4936:readconf.c:372:configlist_print_list:name = h3
...
info:libplutils:4936:readconf.c:372:configlist_print_list:name = libintegral.0.0.0.so
...
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
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 = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:4936:readconf.c:372:configlist_print_list:
      name = /home/ronald/parallelplugins/data/image.input
...
info:libppm:4936:ppm.c:482:ppm_start:mode = 1
info:libppm:4936:ppm.c:614:ppm_load_harnesskernel_replicated:
```

```
      start loading harness kernel and parallel plug-in
info:libppm:4936:ppm.c:1120:ppm_read_integralinput:start reading input file
...
info:libppm:4936:ppm.c:1157:ppm_read_integralinput:input file is opened
info:libppm:4936:ppm.c:1278:ppm_schedule_integral:start scheduling
...
info:libintegral:4892:integral.c:406:integral_init_rmix:start initialization of RMIX
...
info:libintegral:4892:integral.c:441:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:17897:integral.c:202:integral_init:libintegral is starting
...
info:libintegral:17897:integral.c:406:integral_init_rmix:start initialization of RMIX
...
info:libintegral:18076:integral.c:202:integral_init:libintegral is starting
...
info:libintegral:18076:integral.c:406:integral_init_rmix:start initialization of RMIX
...
info:libintegral:18076:integral.c:441:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
info:libintegral:17897:integral.c:441:integral_init_rmix:
      RMIX initialized and integral plug-in exported
...
warn:librmix-rpcx:4936:stream.c:1492:rmix_rpcx_stream_read_reply2:
      unable to read procedure return
warn:librmix-rpcx:4936:rmix-rpcx.c:1662:rmix_rpcx_async_retrieve_job:
      unable to read procedure return
...
info:libintegral:18076:integral.c:649:rmixintegral_integration:integral is -2.772686
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
warn:librmix:4936:rmix.c:1445:rmix_retrieve:unable to invoke method at a remote object
...
warn:1:4936:ppm.c:1086:rmixintegralclient_integration_retrieve:
      unable to invoke remote object method
...
found error, errors = 1
...
info:libppm:4936:ppm.c:1594:ppm_schedule_integral:starting error recovery
...
errors = 1
...
lower = -0.666667
...
upper = 0.666667
...
resend
...
info:libintegral:18076:integral.c:649:rmixintegral_integration:integral is -5.134623
...
info:libppm:4936:ppm.c:1824:ppm_schedule_integral:++++++++++++++++++++++++++++++++++++
info:libppm:4936:ppm.c:1826:ppm_schedule_integral:integral = -10.677033
info:libppm:4936:ppm.c:1828:ppm_schedule_integral:++++++++++++++++++++++++++++++++++++
...
info:libintegral:18076:integral.c:311:integral_fini:libintegral is shutting down
info:libintegral:18076:integral.c:459:integral_fini_rmix:start finalization of RMIX
...
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
...
info:libintegral:18076:integral.c:489:integral_fini_rmix:
```

```
      RMIX finalized and integral plug-in unexported
...
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

```
ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
info:libppm:11632:ppm.c:231:ppm_init:libppm is starting
info:libppm:11632:ppm.c:436:ppm_init_rmix:start initialization of RMIX
...
info:libppm:11632:ppm.c:471:ppm_init_rmix:RMIX initialized
info:libplutils:11632:readconf.c:114:configuration_read_file:start reading configuration file
info:libplutils:11632:readconf.c:388:configlist_print_list:name = h1
info:libplutils:11632:readconf.c:388:configlist_print_list:name = h2
info:libplutils:11632:readconf.c:388:configlist_print_list:name = h3
info:libplutils:11632:readconf.c:388:configlist_print_list:name = joshua
...
info:libplutils:11632:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
...
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
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 = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:11632:readconf.c:388:configlist_print_list:
        name = /home/ronald/parallelplugins/data/image.input
...
info:libppm:11632:ppm.c:570:ppm_start:mode = 2
info:libppm:11632:ppm.c:572:ppm_start:ppmnode = kid
info:libppm:11632:ppm.c:807:ppm_load_harnesskernel_distributed:
      start loading harness kernel and distributed plug-in
...
info:libimgproc:30293:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:30293:imgproc.c:467:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:30293:imgproc.c:500:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libimgproc:21029:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:21029:imgproc.c:467:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:21029:imgproc.c:500:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libimgproc:4666:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:4666:imgproc.c:467:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:4666:imgproc.c:500:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libppm:11632:ppm.c:2459:ppm_read_imageinput:start reading imgproc input file
...
info:libppm:11632:ppm.c:2200:ppm_imageprocessing_loader:
```

```
      source dir /home/ronald/parallelplugins/images
info:libppm:11632:ppm.c:2202:ppm_imageprocessing_loader:
      target dir /home/ronald/parallelplugins/imagetarget
...
info:libimgproc:30293:imgproc.c:2564:imgproc_get_files:3 images found
...
info:libimgproc:30293:imgproc.c:2654:imgproc_loadimages:reading file ngc2237_1024.jpg
info:libimgproc:30293:imgproc.c:2694:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/ngc2237_1024.jpg
...
info:libimgproc:30293:imgproc.c:2864:imgproc_create_magickwand:
      image ngc2237_1024.jpg successfully sent to h2
info:libimgproc:30293:imgproc.c:2654:imgproc_loadimages:reading file image.gif
info:libimgproc:30293:imgproc.c:2694:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/image.gif
...
info:libimgproc:30293:imgproc.c:2864:imgproc_create_magickwand:
      image image.gif successfully sent to h2
info:libimgproc:30293:imgproc.c:2654:imgproc_loadimages:reading file mapimg.gif
info:libimgproc:30293:imgproc.c:2694:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/mapimg.gif
...
info:libimgproc:30293:imgproc.c:2864:imgproc_create_magickwand:
      image mapimg.gif successfully sent to h2
...
info:libimgproc:21029:imgproc.c:1434:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully sent to h3
...
info:libimgproc:4666:imgproc.c:1526:rmiximgproc_passimage:
      acknowledgement for ngc2237_1024.jpg successfully sent to h2
...
info:libimgproc:21029:imgproc.c:1743:rmiximgproc_imageprocessed:
      acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
info:libimgproc:21029:imgproc.c:1434:rmiximgproc_passimage:
      image image.gif successfully sent to h3
...
info:libimgproc:21029:imgproc.c:1434:rmiximgproc_passimage:
      image mapimg.gif successfully sent to h3
...
info:libimgproc:4666:imgproc.c:1526:rmiximgproc_passimage:
      acknowledgement for mapimg.gif successfully sent to h2
...
info:libimgproc:21029:imgproc.c:1743:rmiximgproc_imageprocessed:
      acknowledgement for mapimg.gif successfully sent to h1
...
info:libimgproc:4666:imgproc.c:1526:rmiximgproc_passimage:
      acknowledgement for image.gif successfully sent to h2
...
info:libimgproc:21029:imgproc.c:1743:rmiximgproc_imageprocessed:
      acknowledgement for image.gif successfully sent to h1
...
info:libimgproc:4666:imgproc.c:371:imgproc_fini:libimgproc is shutting down
info:libimgproc:4666:imgproc.c:519:imgproc_fini_rmix:start finalization of RMIX
...
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
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
...
info:libimgproc:4666:imgproc.c:549:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
```

```
info:libimgproc:21029:imgproc.c:549:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
info:libimgproc:30293:imgproc.c:549:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
```

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

```
ronald@kid:~$ harnessd.debug -l libppm.0.0.0.so
...
info:libppm:14502:ppm.c:231:ppm_init:libppm is starting
info:libppm:14502:ppm.c:436:ppm_init_rmix:start initialization of RMIX
...
info:libppm:14502:ppm.c:471:ppm_init_rmix:RMIX initialized
info:libplutils:14502:readconf.c:114:configuration_read_file:
      start reading configuration file
info:libplutils:14502:readconf.c:388:configlist_print_list:name = h1
info:libplutils:14502:readconf.c:388:configlist_print_list:name = h2
info:libplutils:14502:readconf.c:388:configlist_print_list:name = h3
info:libplutils:14502:readconf.c:388:configlist_print_list:name = joshua
...
info:libplutils:14502:readconf.c:388:configlist_print_list:name = libintegral.0.0.0.so
...
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
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 = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:14502:readconf.c:388:configlist_print_list:
      name = /home/ronald/parallelplugins/data/image.input
...
info:libppm:14502:ppm.c:570:ppm_start:mode = 2
info:libppm:14502:ppm.c:572:ppm_start:ppmnode = kid
info:libppm:14502:ppm.c:807:ppm_load_harnesskernel_distributed:
      start loading harness kernel and distributed plug-in
...
info:libimgproc:12137:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:12137:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:12137:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libimgproc:4331:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:4331:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:4331:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libimgproc:23875:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:23875:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:23875:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libppm:14502:ppm.c:2459:ppm_read_imageinput:start reading imgproc input file
info:libppm:14502:ppm.c:2200:ppm_imageprocessing_loader:
      source dir /home/ronald/parallelplugins/images
info:libppm:14502:ppm.c:2202:ppm_imageprocessing_loader:
      target dir /home/ronald/parallelplugins/imagetarget
...
info:libimgproc:12137:imgproc.c:2664:imgproc_get_files:3 images found
```

```
...
info:libimgproc:12137:imgproc.c:2756:imgproc_loadimages:reading file ngc2237_1024.jpg
info:libimgproc:12137:imgproc.c:2796:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/ngc2237_1024.jpg
...
info:libimgproc:12137:imgproc.c:2934:imgproc_create_magickwand:
      image ngc2237_1024.jpg successfully sent to h2
...
info:libppm:14502:ppm.c:2891:rmixppm_repairpipe:***************************************
info:libppm:14502:ppm.c:2893:rmixppm_repairpipe:repair pipe called missing node = h2
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 avalaible
info:libppm:14502:ppm.c:2933:rmixppm_repairpipe:
      name of the plug-in to reload is libimgproc.0.0.0.so
info:libppm:14502:ppm.c:807:ppm_load_harnesskernel_distributed:
      start loading harness kernel and distributed plug-in
...
info:libimgproc:864:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:864:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:864:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libppm:14502:ppm.c:3286:rmixppm_repairpipe:***************************************
...
info:libimgproc:12137:imgproc.c:2756:imgproc_loadimages:reading file image.gif
info:libimgproc:12137:imgproc.c:2796:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/image.gif
info:libimgproc:12137:imgproc.c:3255:imgproc_worklist_printlist:name = ngc2237_1024.jpg
...
info:libimgproc:12137:imgproc.c:2231:rmiximgproc_sendworklist: resend ngc2237_1024.jpg
...
info:libimgproc:12137:imgproc.c:2934:imgproc_create_magickwand:
      image image.gif successfully sent to joshua
info:libimgproc:12137:imgproc.c:2756:imgproc_loadimages:reading file mapimg.gif
info:libimgproc:12137:imgproc.c:2796:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/mapimg.gif
...
info:libimgproc:12137:imgproc.c:2934:imgproc_create_magickwand:
      image mapimg.gif successfully sent to joshua
...
info:libimgproc:4331:imgproc.c:1445:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully sent to h3
...
info:libimgproc:864:imgproc.c:1445:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully sent to h3
info:libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully stored
...
info:libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
      acknowledgement ngc2237_1024.jpg successfully sent to joshua
...
info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
      acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
info:libimgproc:864:imgproc.c:1445:rmiximgproc_passimage:
      image image.gif successfully sent to h3
info:libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully stored
...
info:libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
      acknowledgement ngc2237_1024.jpg successfully sent to joshua
...
```

```
info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
      acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
info:libimgproc:864:imgproc.c:1445:rmiximgproc_passimage:
      image mapimg.gif successfully sent to h3
info:libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
      image mapimg.gif successfully stored
...
info:libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
      acknowledgement mapimg.gif successfully sent to joshua
...
info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
      acknowledgement for mapimg.gif successfully sent to h1
...
info:libimgproc:23875:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
info:libimgproc:23875:imgproc.c:1481:rmiximgproc_passimage:
      image image.gif successfully stored
...
info:libimgproc:23875:imgproc.c:1539:rmiximgproc_passimage:
      acknowledgement image.gif successfully sent to joshua
...
info:libimgproc:864:imgproc.c:1756:rmiximgproc_imageprocessed:
      acknowledgement for image.gif successfully sent to h1
...
info:libimgproc:864:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info:libimgproc:864:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
...
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
...
info:libimgproc:864:imgproc.c:560:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
info:libimgproc:12137:imgproc.c:560:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
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
...
info:libimgproc:4331:imgproc.c:560:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
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
...
info:libppm:14502:ppm.c:519:ppm_fini_rmix:RMIX finalized
...
```

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

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

```
...
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
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 = /home/ronald/parallelplugins/data/montecarlo.input
info:libplutils:14868:readconf.c:388:configlist_print_list:
      name = /home/ronald/parallelplugins/data/image.input
...
info:libppm:14868:ppm.c:570:ppm_start:mode = 2
info:libppm:14868:ppm.c:572:ppm_start:ppmnode = kid
info:libppm:14868:ppm.c:807:ppm_load_harnesskernel_distributed:
      start loading harness kernel and distributed plug-in
...
info:libimgproc:27276:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:27276:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:27276:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libimgproc:20158:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:20158:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:20158:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libimgproc:31774:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:31774:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:31774:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libppm:14868:ppm.c:2459:ppm_read_imageinput:start reading imgproc input file
info:libppm:14868:ppm.c:2474:ppm_read_imageinput:
      more than one input file entries found - try using second one
info:libppm:14868:ppm.c:2200:ppm_imageprocessing_loader:
      source dir /home/ronald/parallelplugins/images
info:libppm:14868:ppm.c:2202:ppm_imageprocessing_loader:
      target dir /home/ronald/parallelplugins/imagetarget
...
info:libimgproc:27276:imgproc.c:2899:imgproc_get_files:3 images found
...
info:libimgproc:27276:imgproc.c:2991:imgproc_loadimages:reading file ngc2237_1024.jpg
info:libimgproc:27276:imgproc.c:3031:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/ngc2237_1024.jpg
...
info:libimgproc:27276:imgproc.c:3237:imgproc_create_magickwand:
      image ngc2237_1024.jpg successfully sent to h2
info:libimgproc:27276:imgproc.c:2991:imgproc_loadimages:reading file image.gif
info:libimgproc:27276:imgproc.c:3031:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/image.gif
...
info:libimgproc:27276:imgproc.c:3237:imgproc_create_magickwand:
      image image.gif successfully sent to h2
info:libimgproc:27276:imgproc.c:2991:imgproc_loadimages:reading file mapimg.gif
info:libimgproc:27276:imgproc.c:3031:imgproc_loadimages:
      full filename is /home/ronald/parallelplugins/images/mapimg.gif
...
info:libimgproc:27276:imgproc.c:3237:imgproc_create_magickwand:
      image mapimg.gif successfully sent to h2
...
info:libimgproc:20158:imgproc.c:1510:rmiximgproc_passimage:
```

```
      image ngc2237_1024.jpg successfully sent to h3
info:libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully stored
warn:librmix-rpcx:27276:stream.c:1492:rmix_rpcx_stream_read_reply2:
      unable to read procedure return
warn:librmix-rpcx:27276:rmix-rpcx.c:1207:rmix_rpcx_oneway_retrieve_job:
      unable to read procedure return
warn:librmix-rpcx:27276:stream.c:1492:rmix_rpcx_stream_read_reply2:
      unable to read procedure return
warn:librmix-rpcx:27276:rmix-rpcx.c:1207:rmix_rpcx_oneway_retrieve_job:
      unable to read procedure return
...
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
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
warn:librmix:31774:rmix.c:1134:rmix_oneway:unable to invoke method at a remote object
...
warn:1:31774:imgproc.c:2058:rmiximgprocclient_imageprocessed_oneway:
      unable to invoke remote object method
warn:libimgproc:31774:imgproc.c:1625:rmiximgproc_passimage:
      could not call remote object for passing image
...
info:libppm:14868:ppm.c:2891:rmixppm_repairpipe:****************************************
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
info:libppm:14868:ppm.c:2908:rmixppm_repairpipe:1 free nodes avalaible
info:libppm:14868:ppm.c:2933:rmixppm_repairpipe:
      name of the plug-in to reload is libimgproc.0.0.0.so
info:libppm:14868:ppm.c:807:ppm_load_harnesskernel_distributed:
      start loading harness kernel and distributed plug-in
...
info:libimgproc:17907:imgproc.c:275:imgproc_init:libimgproc is starting
info:libimgproc:17907:imgproc.c:478:imgproc_init_rmix:start initialization of RMIX
...
info:libimgproc:17907:imgproc.c:511:imgproc_init_rmix:
      RMIX initialized and imgproc plug-in exported
...
info:libppm:14868:ppm.c:3286:rmixppm_repairpipe:****************************************
info:libimgproc:27276:imgproc.c:3548:imgproc_worklist_printlist:name = ngc2237_1024.jpg
info:libimgproc:27276:imgproc.c:3548:imgproc_worklist_printlist:name = image.gif
info:libimgproc:27276:imgproc.c:3548:imgproc_worklist_printlist:name = mapimg.gif
...
info:libimgproc:27276:imgproc.c:2466:rmiximgproc_sendworklistresend ngc2237_1024.jpg
...
info:libimgproc:27276:imgproc.c:2466:rmiximgproc_sendworklist:resend image.gif
...
info:libimgproc:27276:imgproc.c:2466:rmiximgproc_sendworklist:resend mapimg.gif
...
info:libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
      acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
info:libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
      acknowledgement ngc2237_1024.jpg successfully sent to joshua
...
info:libimgproc:17907:imgproc.c:1510:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully sent to h3
info:libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
      image ngc2237_1024.jpg successfully stored
...
info:libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
      acknowledgement ngc2237_1024.jpg successfully sent to joshua
...
```

```
info:libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
      acknowledgement for ngc2237_1024.jpg successfully sent to h1
...
info:libimgproc:17907:imgproc.c:1510:rmiximgproc_passimage:
      image image.gif successfully sent to h3
...
info:libimgproc:17907:imgproc.c:1510:rmiximgproc_passimage:
      image mapimg.gif successfully sent to h3
info:libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
      image mapimg.gif successfully stored
...
info:libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
      acknowledgement mapimg.gif successfully sent to joshua
...
info:libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
      acknowledgement for mapimg.gif successfully sent to h1
info:libimgproc:31774:imgproc.c:1546:rmiximgproc_passimage:
      image image.gif successfully stored
...
info:libimgproc:31774:imgproc.c:1710:rmiximgproc_passimage:
      acknowledgement image.gif successfully sent to joshua
...
info:libimgproc:17907:imgproc.c:1991:rmiximgproc_imageprocessed:
      acknowledgement for image.gif successfully sent to h1
...
info:libimgproc:31774:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info:libimgproc:31774:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
...
info:libimgproc:27276:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info:libimgproc:27276:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
...
info:libimgproc:17907:imgproc.c:382:imgproc_fini:libimgproc is shutting down
info:libimgproc:17907:imgproc.c:530:imgproc_fini_rmix:start finalization of RMIX
...
info:libimgproc:31774:imgproc.c:560:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
info:libimgproc:27276:imgproc.c:560:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
info:libimgproc:17907:imgproc.c:560:imgproc_fini_rmix:
      RMIX finalized and imgproc plug-in unexported
...
```

A.3. Source Listings

A.3.1. RMIX Interface Descriptions

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

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

```
 *
 * - README     for general package information.
 * - INSTALL    for package install information.
 * - COPYING    for package license information and copying conditions.
 * - AUTHORS    for package authors information.
 * - ChangeLog for package changes information.
 *
 * Process the '.in' file with `configure' or `autogen.sh' from the distribution
 * top-level directory to create the target file.
 *
 ***************************************************************************/


/** \file   rmixppm.h
 *  \brief Header file for defining the rmix signatures for the parallel plug-in
 *         manager.
 *
 *  The header file contains the signatures for the communication functions of
 *  the parallel plug-in manger. The signatures define the parameters, which are
 *  sent between the stub functions on each side.
 */


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


/***************************************************************************
 *
 * Macros
 *
 ***************************************************************************/

/* Flag for <harness-rmix.0/harness-rmix.h> header. */
#ifndef     HARNESS_RMIX_HARNESS_RMIX_H
#ifndef     HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
#define     HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H 1
#endif
#endif


/***************************************************************************
 *
 * Method descriptor for repair pipeline method.
 *
 ***************************************************************************/

/** \def    RMIXPPM_REPAIRPIPE_INPUT
 *  \brief Input signature for repair pipeline method.
 */
#define RMIXPPM_REPAIRPIPE_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(string,))           /* name of the node */


/** \def    RMIXPPM_REPAIRPIPE_OUTPUT
 *  \brief Output signature for repair pipeline method.
 */
#define RMIXPPM_REPAIRPIPE_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))              /* return           */

```

```c
/** \def    RMIXPPM_REPAIRPIPE_METHOD
 *  \brief Server-side descriptor for repair pipeline method.
 */
#define RMIXPPM_REPAIRPIPE_METHOD\
    RMIX_METHOD_INIT("repairpipe",\
                     RMIXPPM_REPAIRPIPE_OUTPUT,\
                     RMIXPPM_REPAIRPIPE_INPUT,\
                     rmixppm_repairpipe_call)


/** \def    RMIXPPM_REPAIRPIPE_METHOD_CLIENT
 *  \brief Client-side descriptor for repair pipeline method.
 */
#define RMIXPPM_REPAIRPIPE_METHOD_CLIENT\
    RMIX_METHOD_INIT("repairpipe",\
                     RMIXPPM_REPAIRPIPE_OUTPUT,\
                     RMIXPPM_REPAIRPIPE_INPUT,\
                     NULL)


/*************************************************************************
 *
 * Interface for rmixintegral.
 *
 *************************************************************************/

/** \def    RMIXPPM_METHODS
 *  \brief Server-side method descriptors.
 */
#define RMIXPPM_METHODS \
    { \
        RMIXPPM_REPAIRPIPE_METHOD              /* repairpipe method */\
    }


/** \def    RMIXPPM_METHODS_CLIENT
 *  \brief Client-side method descriptors.
 */
#define RMIXPPM_METHODS_CLIENT \
    { \
        RMIXPPM_REPAIRPIPE_METHOD_CLIENT       /* repairpipe method */\
    }


/** \def    RMIXPPM_METHODS_REPAIRPIPE_INDEX
 *  \brief Method indice repairpipe.
 */
#define RMIXPPM_METHODS_REPAIRPIPE_INDEX 0     /* repairpipe method index */

/** \def    RMIXPPM_METHODS_COUNT
 *  \brief Method indices for rmix repairpipe methods.
 */
#define RMIXPPM_METHODS_COUNT            1     /* method count            */


/*************************************************************************
 *
 * Includes
 *
 *************************************************************************/

/* Include <harness-rmix/harness-rmix.h> header. */
```

```
#if          HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
#include <harness-rmix/harness-rmix.h>
#endif


/**************************************************************************
 *
 * Data
 *
 **************************************************************************/

/** \var    extern const rmix_method_t \
            rmixppm_methods[RMIXPPM_METHODS_COUNT];
 *  \brief Server-side method descriptors for rmix ppm.
 */
extern const rmix_method_t rmixppm_methods[RMIXPPM_METHODS_COUNT];


/** \var    extern const rmix_interface_t rmixppm_interface;
 *  \brief Server-side interface for rmix ppm.
 */
extern const rmix_interface_t rmixppm_interface;


#endif /* RMIXPPM_RMIXPPM_H */


/**************************************************************************
 *
 * END OF FILE
 *
 **************************************************************************/
```

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

```
/**************************************************************************
 *
 * Header file for rmix integral interface.
 * Copyright (c) Ronald Baumann
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
 * top-level directory to create the target file.
 *
 **************************************************************************/

/** \file   rmixintegral.h
 *  \brief Header file for defining the rmix signatures for the integral
 *         communication functions.
 *
 *  The header file contains the signatures for the communication functions of
 *  the Monte Carlo example. The signatures define the parameters, which are
 *  sent between the stub functions on each side.
 */
```

```c

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


/****************************************************************************
 *
 * Macros
 *
 ****************************************************************************/

/* Flag for <harness-rmix.0/harness-rmix.h> header. */
#ifndef     HARNESS_RMIX_HARNESS_RMIX_H
#ifndef     HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
#define     HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H 1
#endif
#endif


/****************************************************************************
 *
 * Method descriptor for integration method.
 *
 ****************************************************************************/

/** \def    RMIXINTEGRAL_INTEGRATION_INPUT
 *  \brief Input signature for integration method.
 */
#define RMIXINTEGRAL_INTEGRATION_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(unsigned_int,            /* iterations      */\
        RMIX_SIGNATURE_TYPE(double,                  /* lower boundary  */\
        RMIX_SIGNATURE_TYPE(double,                  /* upper boundary  */\
        RMIX_SIGNATURE_TYPE(variable_array(double),  /* coefficients    */\
        )))))


/** \def    RMIXINTEGRAL_INTEGRATION_OUTPUT
 *  \brief Output signature for integration method.
 */
#define RMIXINTEGRAL_INTEGRATION_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,                     /* return          */\
        RMIX_SIGNATURE_TYPE(double,)))               /* result          */


/** \def    RMIXINTEGRAL_INTEGRATION_METHOD
 *  \brief Server-side descriptor for integration method.
 */
#define RMIXINTEGRAL_INTEGRATION_METHOD\
    RMIX_METHOD_INIT("integration",\
                     RMIXINTEGRAL_INTEGRATION_OUTPUT,\
                     RMIXINTEGRAL_INTEGRATION_INPUT,\
                     rmixintegral_integration_call)


/** \def    RMIXINTEGRAL_INTEGRATION_METHOD_CLIENT
 *  \brief Client-side descriptor for integration method.
 */
#define RMIXINTEGRAL_INTEGRATION_METHOD_CLIENT\
    RMIX_METHOD_INIT("integration",\
```

```
                              RMIXINTEGRAL_INTEGRATION_OUTPUT, \
                              RMIXINTEGRAL_INTEGRATION_INPUT, \
                              NULL)


/***************************************************************************
 *
 * Interface for rmix integral.
 *
 ***************************************************************************/

/** \def    RMIXINTEGRAL_METHODS
 *  \brief Server-side method descriptors.
 */
#define RMIXINTEGRAL_METHODS \
    { \
        RMIXINTEGRAL_INTEGRATION_METHOD             /* integration method */\
    }


/** \def    RMIXINTEGRAL_METHODS_CLIENT
 *  \brief Client-side method descriptors.
 */
#define RMIXINTEGRAL_METHODS_CLIENT \
    { \
        RMIXINTEGRAL_INTEGRATION_METHOD_CLIENT      /* integration method */\
    }


/** \def    RMIXINTEGRAL_METHODS_INTEGRATION_INDEX
 *  \brief Method indice integration.
 */
#define RMIXINTEGRAL_METHODS_INTEGRATION_INDEX 0   /* integration method index */

/** \def    RMIXINTEGRAL_METHODS_COUNT
 *  \brief Method indices for rmix integration methods.
 */
#define RMIXINTEGRAL_METHODS_COUNT                 1   /* method count             */


/***************************************************************************
 *
 * Includes
 *
 ***************************************************************************/

/* Include <harness-rmix/harness-rmix.h> header. */
#if         HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
#include <harness-rmix/harness-rmix.h>
#endif


/***************************************************************************
 *
 * Data
 *
 ***************************************************************************/

/** \var    extern const rmix_method_t \
            rmixintegral_methods[RMIXINTEGRAL_METHODS_COUNT];
 *  \brief Server-side method descriptors for rmix integral.
 */
extern const rmix_method_t rmixintegral_methods[RMIXINTEGRAL_METHODS_COUNT];
```

```c

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


#endif /* RMIXINTEGRAL_RMIXINTEGRAL_H */

/***************************************************************************
 *
 * END OF FILE
 *
 ***************************************************************************/
```

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

```c
/***************************************************************************
 *
 * Header file for rmix image processing pipeline interface.
 * Copyright (c) Ronald Baumann
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
 * top-level directory to create the target file.
 *
 ***************************************************************************/

/** \file  rmiximgproc.h
 *  \brief Header file for defining the rmix signatures for the image
 *         processing pipeline.
 *
 *  The header file contains the signatures for the communication functions of
 *  the image processing example. The signatures define the parameters, which
 *  are sent between the stub functions on each side.
 */


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


/***************************************************************************
 *
 * Macros
 *
 ***************************************************************************/

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

```
#endif
#endif


/****************************************************************************
 *
 * Method descriptor for initlisation of the pipeline method.
 *
 ****************************************************************************/

/** \def    RMIXIMGPROC_INITPIPELINE_INPUT
 *  \brief Input signature for initpipeline method.
 */
#define RMIXIMGPROC_INITPIPELINE_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(unsigned_int,        /* filter selection  */\
        RMIX_SIGNATURE_TYPE(string,              /* source dir        */\
        RMIX_SIGNATURE_TYPE(string,              /* traget dir        */\
        RMIX_SIGNATURE_TYPE(string,              /* successor         */\
        RMIX_SIGNATURE_TYPE(string,              /* predecessor       */\
        RMIX_SIGNATURE_TYPE(string,              /* ppm node          */\
        )))))))

/** \def    RMIXIMGPROC_INITPIPELINE_OUTPUT
 *  \brief Output signature for initpipeline method.
 */
#define RMIXIMGPROC_INITPIPELINE_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                /* return            */

/** \def    RMIXIMGPROC_INITPIPELINE_METHOD
 *  \brief Server-side descriptor for initpipeline method.
 */
#define RMIXIMGPROC_INITPIPELINE_METHOD\
    RMIX_METHOD_INIT("initpipeline",\
                     RMIXIMGPROC_INITPIPELINE_OUTPUT,\
                     RMIXIMGPROC_INITPIPELINE_INPUT,\
                     rmiximgproc_initpipeline_call)

/** \def    RMIXIMGPROC_INITPIPELINE_METHOD
 *  \brief Client-side descriptor for initpipeline method.
 */
#define RMIXIMGPROC_INITPIPELINE_METHOD_CLIENT\
    RMIX_METHOD_INIT("initpipeline",\
                     RMIXIMGPROC_INITPIPELINE_OUTPUT,\
                     RMIXIMGPROC_INITPIPELINE_INPUT,\
                     NULL)

/****************************************************************************
 *
 * Method descriptor for invokepipeline method.
 *
 ****************************************************************************/

/** \def    RMIXIMGPROC_INVOKEPIPELINE_INPUT
 *  \brief Input signature for invokepipeline method.
 */
#define RMIXIMGPROC_INVOKEPIPELINE_INPUT\
    RMIX_SIGNATURE()                              /* void   */
```

```
/** \def    RMIXIMGPROC_INVOKEPIPELINE_OUTPUT
 *  \brief Output signature for invokepipeline method.
 */
#define RMIXIMGPROC_INVOKEPIPELINE_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                    /* return */

/** \def    RMIXIMGPROC_INVOKEPIPELINE_METHOD
 *  \brief Server-side descriptor for invokepipeline method.
 */
#define RMIXIMGPROC_INVOKEPIPELINE_METHOD\
    RMIX_METHOD_INIT("invokepipeline",\
                     RMIXIMGPROC_INVOKEPIPELINE_OUTPUT,\
                     RMIXIMGPROC_INVOKEPIPELINE_INPUT,\
                     rmiximgproc_invokepipeline_call)

/** \def    RMIXIMGPROC_INVOKEPIPELINE_METHOD_CLIENT
 *  \brief Client-side descriptor for invokepipeline method.
 */
#define RMIXIMGPROC_INVOKEPIPELINE_METHOD_CLIENT\
    RMIX_METHOD_INIT("invokepipeline",\
                     RMIXIMGPROC_INVOKEPIPELINE_OUTPUT,\
                     RMIXIMGPROC_INVOKEPIPELINE_INPUT,\
                     NULL)

/******************************************************************************
 *
 * Method descriptor for send backup list method.
 *
 ******************************************************************************/

/** \def    RMIXIMGPROC_SENDWORKLIST_INPUT
 *  \brief Input signature for send backup list method.
 */
#define RMIXIMGPROC_SENDWORKLIST_INPUT\
    RMIX_SIGNATURE()                                  /* void   */

/** \def    RMIXIMGPROC_SENDWORKLIST_OUTPUT
 *  \brief Output signature for send backup list method.
 */
#define RMIXIMGPROC_SENDWORKLIST_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                    /* return */

/** \def    RMIXIMGPROC_SENDWORKLIST_METHOD
 *  \brief Server-side descriptor for send back up list method.
 */
#define RMIXIMGPROC_SENDWORKLIST_METHOD\
    RMIX_METHOD_INIT("sendworklist",\
                     RMIXIMGPROC_SENDWORKLIST_OUTPUT,\
                     RMIXIMGPROC_SENDWORKLIST_INPUT,\
                     rmiximgproc_sendworklist_call)

/** \def    RMIXIMGPROC_SENDWORKLIST_METHOD_CLIENT
 *  \brief Client-side descriptor for send backup list method.
```

```
 */
#define RMIXIMGPROC_SENDWORKLIST_METHOD_CLIENT\
    RMIX_METHOD_INIT("sendworklist",\
                     RMIXIMGPROC_SENDWORKLIST_OUTPUT,\
                     RMIXIMGPROC_SENDWORKLIST_INPUT,\
                     NULL)


/***************************************************************************
 *
 * Method descriptor for availability check method.
 *
 ***************************************************************************/

/** \def   RMIXIMGPROC_AVAILABILITYCHECK_INPUT
 *  \brief Input signature for availability check method.
 */
#define RMIXIMGPROC_AVAILABILITYCHECK_INPUT\
    RMIX_SIGNATURE()                                  /* void   */


/** \def   RMIXIMGPROC_AVAILABILITYCHECK_OUTPUT
 *  \brief Output signature for availability check method.
 */
#define RMIXIMGPROC_AVAILABILITYCHECK_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                    /* return */


/** \def   RMIXIMGPROC_AVAILABILITYCHECK_METHOD
 *  \brief Server-side descriptor for availability check method.
 */
#define RMIXIMGPROC_AVAILABILITYCHECK_METHOD\
    RMIX_METHOD_INIT("availabilitycheck",\
                     RMIXIMGPROC_AVAILABILITYCHECK_OUTPUT,\
                     RMIXIMGPROC_AVAILABILITYCHECK_INPUT,\
                     rmiximgproc_availabilitycheck_call)


/** \def   RMIXIMGPROC_AVAILABILITYCHECK_METHOD_CLIENT
 *  \brief Client-side descriptor for availability check method.
 */
#define RMIXIMGPROC_AVAILABILITYCHECK_METHOD_CLIENT\
    RMIX_METHOD_INIT("availabilitycheck",\
                     RMIXIMGPROC_AVAILABILITYCHECK_OUTPUT,\
                     RMIXIMGPROC_AVAILABILITYCHECK_INPUT,\
                     NULL)


/***************************************************************************
 *
 * Method descriptor for passimage method.
 *
 ***************************************************************************/

/** \def   RMIXIMGPROC_PASSIMAGE_INPUT
 *  \brief Input signature for pass image method.
 */
#define RMIXIMGPROC_PASSIMAGE_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(string,                          /* image name */\
        RMIX_SIGNATURE_TYPE(variable_array(unsigned_char), /* image blob */\
        )))
```

```
/** \def    RMIXIMGPROC_PASSIMAGE_OUTPUT
 *  \brief Output signature for pass image method.
 */
#define RMIXIMGPROC_PASSIMAGE_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                         /* return       */

/** \def    RMIXIMGPROC_PASSIMAGE_METHOD
 *  \brief Server-side descriptor for pass image method.
 */
#define RMIXIMGPROC_PASSIMAGE_METHOD\
    RMIX_METHOD_INIT("passimage",\
                     RMIXIMGPROC_PASSIMAGE_OUTPUT,\
                     RMIXIMGPROC_PASSIMAGE_INPUT,\
                     rmiximgproc_passimage_call)

/** \def    RMIXIMGPROC_PASSIMAGE_METHOD_CLIENT
 *  \brief Client-side descriptor for pass image method.
 */
#define RMIXIMGPROC_PASSIMAGE_METHOD_CLIENT\
    RMIX_METHOD_INIT("passimage",\
                     RMIXIMGPROC_PASSIMAGE_OUTPUT,\
                     RMIXIMGPROC_PASSIMAGE_INPUT,\
                     NULL)

/****************************************************************************
 *
 * Method descriptor for set image counter method.
 *
 ****************************************************************************/

/** \def    RMIXIMGPROC_SETIMAGECOUNTER_INPUT
 *  \brief Input signature for set image counter method.
 */
#define RMIXIMGPROC_SETIMAGECOUNTER_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(unsigned_int,))        /* counter  */

/** \def    RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT
 *  \brief Output signature for set image counter method.
 */
#define RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                 /* return   */

/** \def    RMIXIMGPROC_SETIMAGECOUNTER_METHOD
 *  \brief Server-side descriptor for set image counter method.
 */
#define RMIXIMGPROC_SETIMAGECOUNTER_METHOD\
    RMIX_METHOD_INIT("setimagecounter",\
                     RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT,\
                     RMIXIMGPROC_SETIMAGECOUNTER_INPUT,\
                     rmiximgproc_setimagecounter_call)

/** \def    RMIXIMGPROC_SETIMAGECOUNTER_METHOD_CLIENT
```

```
 *  \brief Client-side descriptor for set image counter method.
 */
#define RMIXIMGPROC_SETIMAGECOUNTER_METHOD_CLIENT\
    RMIX_METHOD_INIT("setimagecounter",\
                     RMIXIMGPROC_SETIMAGECOUNTER_OUTPUT,\
                     RMIXIMGPROC_SETIMAGECOUNTER_INPUT,\
                     NULL)


/*****************************************************************************
 *
 * Method descriptor for update image counter method.
 *
 *****************************************************************************/

/** \def  RMIXIMGPROC_UPDATEIMAGECOUNTER_INPUT
 *  \brief Input signature for update image counter method.
 */
#define RMIXIMGPROC_UPDATEIMAGECOUNTER_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(unsigned_int,))          /* counter  */

/** \def  RMIXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT
 *  \brief Output signature for update image counter method.
 */
#define RMIXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                   /* return   */

/** \def  RMIXIMGPROC_UPDATEIMAGECOUNTER_METHOD
 *  \brief Server-side descriptor for update image counter method.
 */
#define RMIXIMGPROC_UPDATEIMAGECOUNTER_METHOD\
    RMIX_METHOD_INIT("updateimagecounter",\
                     RMIXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT,\
                     RMIXIMGPROC_UPDATEIMAGECOUNTER_INPUT,\
                     rmiximgproc_setimagecounter_call)

/** \def  RMIXIMGPROC_UPDATEIMAGECOUNTER_METHOD_CLIENT
 *  \brief Client-side descriptor for update image counter method.
 */
#define RMIXIMGPROC_UPDATEIMAGECOUNTER_METHOD_CLIENT\
    RMIX_METHOD_INIT("updateimagecounter",\
                     RMIXIMGPROC_UPDATEIMAGECOUNTER_OUTPUT,\
                     RMIXIMGPROC_UPDATEIMAGECOUNTER_INPUT,\
                     NULL)


/*****************************************************************************
 *
 * Method descriptor for update predecessor method.
 *
 *****************************************************************************/

/** \def  RMIXIMGPROC_UPDATEPREDECESSOR_INPUT
 *  \brief Input signature for update predecessor method.
 */
#define RMIXIMGPROC_UPDATEPREDECESSOR_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(string, ))             /* successor      */
```

```

/** \def    RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT
 *  \brief Output signature for update predecessor method.
 */
#define RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,                  /* return          */\
        RMIX_SIGNATURE_TYPE(unsigned_int,)))      /* image counter   */

/** \def    RMIXIMGPROC_UPDATEPREDECESSOR_METHOD
 *  \brief Server-side descriptor for update predecessor method.
 */
#define RMIXIMGPROC_UPDATEPREDECESSOR_METHOD\
    RMIX_METHOD_INIT("updatepredecessor",\
                     RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT,\
                     RMIXIMGPROC_UPDATEPREDECESSOR_INPUT,\
                     rmiximgproc_updatepredecessor_call)

/** \def    RMIXIMGPROC_UPDATEPREDECESSOR_METHOD_CLIENT
 *  \brief Client-side descriptor for update predecessor method.
 */
#define RMIXIMGPROC_UPDATEPREDECESSOR_METHOD_CLIENT\
    RMIX_METHOD_INIT("updatepredecessor",\
                     RMIXIMGPROC_UPDATEPREDECESSOR_OUTPUT,\
                     RMIXIMGPROC_UPDATEPREDECESSOR_INPUT,\
                     NULL)

/******************************************************************************
 *
 * Method descriptor for update successor method.
 *
 ******************************************************************************/

/** \def    RMIXIMGPROC_UPDATESUCCESSOR_INPUT
 *  \brief Input signature for update successor method.
 */
#define RMIXIMGPROC_UPDATESUCCESSOR_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(string, ))            /* predecessor      */

/** \def    RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT
 *  \brief Output signature for update successor method.
 */
#define RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int, ))               /* return          */

/** \def    RMIXIMGPROC_UPDATESUCCESSOR_METHOD
 *  \brief Server-side descriptor for update successor method.
 */
#define RMIXIMGPROC_UPDATESUCCESSOR_METHOD\
    RMIX_METHOD_INIT("updatesuccessor",\
                     RMIXIMGPROC_UPDATESUCCESSOR_OUTPUT,\
                     RMIXIMGPROC_UPDATESUCCESSOR_INPUT,\
                     rmiximgproc_updatesuccessor_call)

```

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


/**************************************************************************
 *
 * Method descriptor for imageprocessed.
 *
 **************************************************************************/

/** \def     RMIXIMGPROC_IMAGEPROCESSED_INPUT
 *  \brief Input signature for image processed method.
 */
#define RMIXIMGPROC_IMAGEPROCESSED_INPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(string, ))                  /* image name */


/** \def     RMIXIMGPROC_IMAGEPROCESSED_OUTPUT
 *  \brief Output signature for image processed method.
 */
#define RMIXIMGPROC_IMAGEPROCESSED_OUTPUT\
    RMIX_SIGNATURE(\
        RMIX_SIGNATURE_TYPE(int,))                      /* return      */


/** \def     RMIXIMGPROC_IMAGEPROCESSED_METHOD
 *  \brief Server-side descriptor for image processed method.
 */
#define RMIXIMGPROC_IMAGEPROCESSED_METHOD\
    RMIX_METHOD_INIT("imageprocessed",\
                     RMIXIMGPROC_IMAGEPROCESSED_OUTPUT,\
                     RMIXIMGPROC_IMAGEPROCESSED_INPUT,\
                     rmiximgproc_imageprocessed_call)


/** \def     RMIXIMGPROC_IMAGEPROCESSED_METHOD_CLIENT
 *  \brief Client-side descriptor for image processed method.
 */
#define RMIXIMGPROC_IMAGEPROCESSED_METHOD_CLIENT\
    RMIX_METHOD_INIT("imageprocessed",\
                     RMIXIMGPROC_IMAGEPROCESSED_OUTPUT,\
                     RMIXIMGPROC_IMAGEPROCESSED_INPUT,\
                     NULL)


/**************************************************************************
 *
 * Interface for rmix imgproc.
 *
 **************************************************************************/

/** \def     RMIXIMGPROC_METHODS
 *  \brief Server-side method descriptors.
 */
#define RMIXIMGPROC_METHODS \
    { \
```

```
        RMIXIMGPROC_INITPIPELINE_METHOD,          /* init pipeline method      */\
        RMIXIMGPROC_INVOKEPIPELINE_METHOD,        /* invoke pipeline method    */\
        RMIXIMGPROC_PASSIMAGE_METHOD,             /* pass image method         */\
        RMIXIMGPROC_SETIMAGECOUNTER_METHOD,       /* set image counter method */\
        RMIXIMGPROC_AVAILABILITYCHECK_METHOD,     /* check plug-in existence   */\
        RMIXIMGPROC_IMAGEPROCESSED_METHOD,        /* image processed           */\
        RMIXIMGPROC_UPDATEPREDECESSOR_METHOD,     /* update predecessor        */\
        RMIXIMGPROC_UPDATESUCCESSOR_METHOD,       /* update successor          */\
        RMIXIMGPROC_SETIMAGECOUNTER_METHOD,       /* update image counter      */\
        RMIXIMGPROC_SENDWORKLIST_METHOD           /* send work list method     */\
    }


/** \def    RMIXIMGPROC_METHODS_CLIENT
 *  \brief Client-side method descriptors.
 */
#define RMIXIMGPROC_METHODS_CLIENT \
    { \
        RMIXIMGPROC_INITPIPELINE_METHOD_CLIENT,     /* init pipeline method   */\
        RMIXIMGPROC_INVOKEPIPELINE_METHOD_CLIENT,   /* invoke pipeline method */\
        RMIXIMGPROC_PASSIMAGE_METHOD_CLIENT,        /* pass image method      */\
        RMIXIMGPROC_SETIMAGECOUNTER_METHOD_CLIENT,/* set iamge count method */\
        RMIXIMGPROC_AVAILABILITYCHECK_METHOD_CLIENT, /* check existence       */\
        RMIXIMGPROC_IMAGEPROCESSED_METHOD_CLIENT,    /* image processed       */\
        RMIXIMGPROC_UPDATEPREDECESSOR_METHOD_CLIENT, /* update predecessor    */\
        RMIXIMGPROC_UPDATESUCCESSOR_METHOD_CLIENT,   /* update successor      */\
        RMIXIMGPROC_UPDATEIMAGECOUNTER_METHOD_CLIENT,/* set iamge counter     */\
        RMIXIMGPROC_SENDWORKLIST_METHOD_CLIENT       /* send work list        */\
    }


/** \def    RMIXIMGPROC_METHODS_INITPIPELINE_INDEX
 *  \brief Method indice initpipeline.
 */
#define RMIXIMGPROC_METHODS_INITPIPELINE_INDEX        0 /* init pipe index   */

/** \def    RMIXIMGPROC_METHODS_INVOKEPIPELINE_INDEX
 *  \brief Method indice invokepipeline.
 */
#define RMIXIMGPROC_METHODS_INVOKEPIPELINE_INDEX      1 /* invoke pipe index */

/** \def    RMIXIMGPROC_METHODS_PASSIMAGE_INDEX
 *  \brief Method indice passimage.
 */
#define RMIXIMGPROC_METHODS_PASSIMAGE_INDEX           2 /* pass image index  */

/** \def    RMIXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX
 *  \brief Method indice setimagecounter.
 */
#define RMIXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX     3 /* set image counter */

/** \def    RMIXIMGPROC_METHODS_AVAILABILITYCHECK_INDEX
 *  \brief Method indice availabilitycheck.
 */
#define RMIXIMGPROC_METHODS_AVAILABILITYCHECK_INDEX   4 /* check existence   */

/** \def    RMIXIMGPROC_METHODS_IMAGEPROCESSED_INDEX
 *  \brief Method indice imageprocessed.
 */
#define RMIXIMGPROC_METHODS_IMAGEPROCESSED_INDEX      5 /* image processed   */

/** \def    RMIXIMGPROC_METHODS_UPDATEPREDECESSOR_INDEX
 *  \brief Method indice update predecessor.
```

```
 */
#define RMIXIMGPROC_METHODS_UPDATEPREDECESSOR_INDEX  6 /* updatepredecessor */

/** \def    RMIXIMGPROC_METHODS_UPDATESUCCESSOR_INDEX
 *  \brief Method indice update successor.
 */
#define RMIXIMGPROC_METHODS_UPDATESUCCESSOR_INDEX    7 /* update successor  */

/** \def    RMIXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX
 *  \brief Method indice updateimagecounter.
 */
#define RMIXIMGPROC_METHODS_UPDATEIMAGECOUNTER_INDEX 8 /* update counter    */

/** \def    RMIXIMGPROC_METHODS_SENDWORKLIST_INDEX
 *  \brief Method indice send worklist.
 */
#define RMIXIMGPROC_METHODS_SENDWORKLIST_INDEX       9 /* send worklist     */

/** \def    RMIXIMGPROC_METHODS_COUNT
 *  \brief Method indices for rmix image processing methods.
 */
#define RMIXIMGPROC_METHODS_COUNT                   10 /* method count      */


/****************************************************************************
 *
 * Includes
 *
 ****************************************************************************/

/* Include <harness-rmix/harness-rmix.h> header. */
#if         HAVE_HARNESS_RMIX_0_HARNESS_RMIX_H
#include <harness-rmix/harness-rmix.h>
#endif


/****************************************************************************
 *
 * Data
 *
 ****************************************************************************/

/** \var    extern const rmix_method_t \
            rmiximgproc_methods[RMIXIMGPROC_METHODS_COUNT];
 *  \brief Server-side method descriptors for rmix imgproc.
 */
extern const rmix_method_t rmiximgproc_methods[RMIXIMGPROC_METHODS_COUNT];


/** \var    extern const rmix_interface_t rmiximgproc_interface;
 *  \brief Server-side interface for rmix imgproc.
 */
extern const rmix_interface_t rmiximgproc_interface;

#endif /* RMIXIMGPROC_RMIXIMGPROC_H */

/****************************************************************************
 *
 * END OF FILE
 *
 ****************************************************************************/
```

A.3.2. Parallel Plug-in Manager

A.3.2.1. Header File for the PPM

```
/* libppm/ppm.h.  Generated by configure.  */
/*************************************************************************
 *
 * Header file for the parallel plug-in manager module.
 * Copyright (c) Ronald Baumann
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README     for general package information.
 * - INSTALL    for package install information.
 * - COPYING    for package license information and copying conditions.
 * - AUTHORS    for package authors information.
 * - ChangeLog  for package changes information.
 *
 * Process the '.in' file with 'configure' or 'autogen.sh' from the distribution
 * top-level directory to create the target file.
 *
 ************************************************************************/

/** \file ppm.h
 *  \brief Header file for parallel plug-in manager module.
 *
 *  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
 *  of harness daemons.
 */

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


/************************************************************************
 *
 * Macros
 *
 ************************************************************************/

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

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

/** \def    LIBPPM_PACKAGE_RELEASE
 *  \brief Package release (unquoted).
 */
#define LIBPPM_PACKAGE_RELEASE 0
```

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

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

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

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

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

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

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

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

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

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

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

```
#define PPM_WARN(string) \
  fprintf(stderr,\
          PPM_STRING(warn:libppm:%d:%s:%u:%s:string\n),\
          getpid(), __FILE__, __LINE__, __FUNC__


/** \def   PPM_INFO(string)
 *  \brief Debug printout.
 */
#define PPM_INFO(string) \
  fprintf(stderr,\
          PPM_STRING(info:libppm:%d:%s:%u:%s:string\n),\
          getpid(), __FILE__, __LINE__, __FUNC__


/** \def   PPM_PRINT(string)
 *  \brief Wrapper for debug printout.
 */
#ifdef      DEBUG
#define     PPM_PRINT(string) string);
#else   /* DEBUG */
#define     PPM_PRINT(string)
#endif /* DEBUG */


/* Flag for <harness.0/harness.h> header. */
#ifndef      HARNESS_HARNESS_H
#ifndef      HAVE_HARNESS_0_HARNESS_H
#define       HAVE_HARNESS_0_HARNESS_H 1
#endif  /* HAVE_HARNESS_0_HARNESS_H */
#endif  /* HARNESS_HARNESS_H */


/** \def   FREE(x)
 *  \brief Macro frees allocated memory and add the NULL pointer.
 *
 *  First the pointer is checked whether it is not NULL. Then it is freed.
 */
#define FREE(x) {if (x != NULL) {free(x); x = NULL;}}


/** \def   SOURCEDIR
 *  \brief Dirs entry in the imgproc input file.
 */
#define SOURCEDIR "sourcedir"


/** \def   TARGETDIR
 *  \brief Target directory entry in the imgproc input file.
 */
#define TARGETDIR "targetdir"


/****************************************************************************
 *
 * Includes
 *
 ****************************************************************************/

/* Include <harness.0/harness.h> header. */
#if         HAVE_HARNESS_0_HARNESS_H
#include        <harness.0/harness.h>
#endif /* HAVE_HARNESS_0_HARNESS_H */
```

```c

#include "readconf.h"
#include "rmixintegral.h"
#include "rmiximgproc.h"
#include "rmixppm.h"


/*****************************************************************************
 *
 * Data Types
 *
 *****************************************************************************/

/** \struct ppm_t
 *  \brief Parallel Plug-in Manager "class".
 *
 *  This structure contains the public plug-in data and function pointers.
 *  It also provides information about the version of the plug-in.
 */
typedef struct
{
    struct
    {
        const unsigned int current;     /* current version        */
        const unsigned int revision;    /* current revision       */
        const unsigned int age;         /* version age            */
        const unsigned int first;       /* first supported version */
    }version;                           /* plug-in version        */
}ppm_t;


/*****************************************************************************
 *
 * Function Prototypes
 *
 *****************************************************************************/

/** \fn     int ppm_init_rmix();
 *  \brief  Exports and initialises the RMIX library.
 *
 *  \return 0 on success or -1 on any error
 */
int ppm_init_rmix();


/*  \fn     int ppm_fini_rmix();
 *  \brief  Finalizes and unexports RMIX.
 *
 *  \return 0 on success or -1 on any error
 */
int ppm_fini_rmix();


/** \fn     int ppm_start()
 *  \brief  Invokes the Parallel Plug-in Manager execution.
 *
 *  \return 0 on success or -1 on any error
 */
int ppm_start();


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

```
 *  \brief  Loads a replicated parallel plug-in on the Harness kernels specified
 *          in the conf file.
 *
 *  \param  *nodelist   Name of the node(s)
 *  \param  *pluginlist Name of the parallel plug-in
 *  \return             0 on success or -1 on any error
 */
int ppm_load_harnesskernel_replicated( configlist_t *nodelist,
                                       configlist_t *pluginlist);


/** \fn     int ppm_load_harnesskernel_distributed( configlist_t *nodelist, \
                                                   configlist_t *pluginlist, \
                                                   char         *ppmnode);
 *  \brief  Loads a distributed parallel plug-in on the Harness kernels
 *          specified in the conf file.
 *
 *  \param  **nodelist   Name of the node(s)
 *  \param   *pluginlist Name of the parallel plug-in
 *  \param   *ppmnode    Node running the PPM
 *  \return              0 on success or -1 on any error
 */
int ppm_load_harnesskernel_distributed( configlist_t **nodelist,
                                        configlist_t  *pluginlist,
                                        char          *ppmnode);


/** \fn     int ppm_unload_harnesskernel( configlist_t *nodelist);
 *  \brief  Unoads the Harness kernels on the nodes specified in the conf file.
 *          The kernels are completely killed inclusive all running plug-ins.
 *          Careful use of function is advised.
 *
 *  \param  *nodelist   Name of the node(s)
 *  \return             0 on success or -1 on any error
 */
int ppm_unload_harnesskernel( configlist_t *nodelist);


/** \fn     int ppm_unload_harnesskernel_soft( configlist_t *nodelist);
 *  \brief  Unoads the Harness kernels on the nodes specified in the conf file.
 *          The kernels are completely killed inclusive all running plug-ins.
 *          For terminating the kernel the internal shutdown function is used.
 *
 *  \param  *nodelist   Name of the node(s)
 *  \return             0 on success or -1 on any error
 */
int ppm_unload_harnesskernel_soft( configlist_t *nodelist);


/** \fn     void ppm_sleep(int sec_dlay, int usec_dlay);
 *  \brief  The process sleeps a preset time interval.
 *
 *  \param   sec_dlay   Delay in seconds
 *  \param  usec_dlay   Delay in micro seconds
 */
void ppm_sleep(int sec_dlay, int usec_dlay);


/** \fn     int ppm_createremoteref( rmix_remoteref_t **remoteobj,\
                                   char              *objectid,\
                                   char              *node);
 *  \brief  Creates a remote reference specified by objectid and node. The RPC
 *          protocol is used.
```

```
 *
 * \param    **remoteobj  ID of the remote object
 * \param     *objectid   ID of the exported object
 * \param     *node       Name of the node
 * \return                0 on success or -1 on any error
 */
int ppm_createremoteref( rmix_remoteref_t **remoteobj,
                         char               *objectid,
                         char               *node);


/*****************************************************************************
 *
 * Monte Carlo section
 *
 *****************************************************************************/

/** \fn     int ppm_monte_carlo_loader( configlist_t *nodelist, \
                                       configlist_t *replicatedlist, \
                                       configlist_t *inputlist)
 * \brief  Invokes the Monte Carlo parallel plug-in execution.
 *
 * \param  *nodelist         Name of the node(s)
 * \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
 */
int ppm_monte_carlo_loader( configlist_t *nodelist,
                            configlist_t *replicatedlist,
                            configlist_t *inputlist);


/** \fn     int rmixintegralclient_integration_send(                     \
                                       rmix_invokeref_t **invokeref,    \
                                       rmix_remoteref_t  *remoteobject, \
                                       unsigned int       iterations,   \
                                       double             lowerbound,   \
                                       double             upperbound,   \
                                       unsigned int       numcoeffs,    \
                                       double            *coeffs);
 * \brief  Sends an integral dataset to a server (client-side method stub).
 *
 * \param  **invokeref     The invocation reference (also return).
 * \param   *remoteobject  The remote object reference.
 * \param    iterations    The number of the generated random numbers.
 * \param    lowerbound    The lower boundary of the integration interval.
 * \param    upperbound    The upper boundary of the integration interval.
 * \param    numcoeffs     The number of function coefficients.
 * \param   *coeffs        The coefficients.
 * \return                 0 on success or -1 on any error.
 */
int rmixintegralclient_integration_send( rmix_invokeref_t **invokeref,
                                         rmix_remoteref_t  *remoteobject,
                                         unsigned int       iterations,
                                         double             lowerbound,
                                         double             upperbound,
                                         unsigned int       numcoeffs,
                                         double            *coeffs);


/** \fn     int rmixintegralclient_integration_retrieve(               \
                                      rmix_invokeref_t **invokeref, \
                                      double            *integral);
```

```
 *  \brief  Retrieves rmixintegral_integration invocation result (client-side
 *          method stub).
 *
 *  Returns the output of a previous asynchonous invocation that sends the
 *  integration parameters to the server and starts the integration
 *  process.
 *
 *  \param  **invokeref   The invocation reference.
 *  \param   *integral    The result of the integration.
 *  \return               0 on success or -1 on any error.
 */
int rmixintegralclient_integration_retrieve( rmix_invokeref_t **invokeref,
                                             double            *integral);


/** \fn     int ppm_read_integralinput( configlist_t  *inputlist, \
                                       unsigned int  *number, \
                                       double       **coeffs, \
                                       int           *iterations, \
                                       double        *lower, \
                                       double *upper);
 *  \brief  Reads the input file, which contains the needed information for
 *          the integral computation.
 *
 *  \param   *inputlist    List with name(s) of input file(s)
 *  \param   *number       Returns the number of coefficients
 *  \param  **coeffs       Returns the array of coefficients
 *  \param   *iterations   Returns the amount of random numbers, which are used
 *  \param   *lower        Returns the lower boundary of the interval
 *  \param   *upper        Returns the upper boundary of the interval
 *  \return                0 on success or -1 on any error
 */
int ppm_read_integralinput( configlist_t  *inputlist,
                            unsigned int  *number,
                            double       **coeffs,
                            int           *iterations,
                            double        *lower,
                            double        *upper);


/** \fn     int ppm_schedule_integral( configlist_t *nodelist, \
                                      configlist_t *pluginlist, \
                                      unsigned int  numcoeffs, \
                                      double       *coeffs, \
                                      int           iterations, \
                                      double        lower_border, \
                                      double        upper_border);
 *  \brief  Partitions the integration interval and sends it to the parallel
 *          plug-in units.
 *
 *  \param  *nodelist     Name of the node(s)
 *  \param  *pluginlist   Name of the parallel plug-in
 *  \param  numcoeffs     Number of coefficients
 *  \param  *coeffs       Array of coefficients
 *  \param  iterations    Amount of random numbers, which are used
 *  \param  lower_border  Lower boundary of the interval
 *  \param  upper_border  Upper boundary of the interval
 *  \return               0 on success or -1 on any error
 */
int ppm_schedule_integral( configlist_t *nodelist,
                           configlist_t *plugin_list,
                           unsigned int  numcoeffs,
```

```
                           double       *coeffs ,
                           int           iterations ,
                           double        lower_border ,
                           double        upper_border );


/***************************************************************************
 *
 * Image Processing section
 *
 ***************************************************************************/

/** \fn     int rmiximgprocclient_initpipeline (rmix_remoteref_t *remoteobj , \
                                               unsigned int       filter , \
                                               char              *sourcedir , \
                                               char              *targetdir , \
                                               char              *successor , \
                                               char              *predecessor , \
                                               char              *ppmnode );
 *  \brief  Initialises the pipeline (client-side method stub).
 *
 *  \param  *remoteobj    ID of the remote object
 *  \param   filter       Image filter which will be used
 *  \param  *sourcedir    Directory containing image sources.
 *  \param  *targetdir    Directory containing processed images
 *  \param  *successor    ID of the next plug-in in the pipeline
 *  \param  *predecessor  ID of the previous plug-in in the pipeline
 *  \param  *ppmnode      Node, on which the PPM runs
 *  \return               0 on success or -1 on any error
 */
int rmiximgprocclient_initpipeline ( rmix_remoteref_t *remoteobj ,
                                     unsigned int      filter ,
                                     char             *sourcedir ,
                                     char             *targetdir ,
                                     char             *successor ,
                                     char             *predecessor ,
                                     char             *ppmnode );


/** \fn     int rmiximgprocclient_invokepipeline_oneway ( \
                                            rmix_remoteref_t *remoteobj );
 *  \brief  Invokes the pipeline processing (client-side method stub).
 *
 *  \param  *remoteobj  ID of the remote object
 *  \return             0 on success or -1 on any error
 */
int rmiximgprocclient_invokepipeline_oneway (rmix_remoteref_t *remoteobj );


/** \fn     int rmiximgprocclient_availabilitycheck( \
                                    rmix_remoteref_t *remoteobj );
 *  \brief  Checks the availability of a plug-in (client-side method stub).
 *
 *  \param  *remoteobj  ID of the remote object
 *  \return             0 on success or -1 on any error
 */
int rmiximgprocclient_availabilitycheck( rmix_remoteref_t *remoteobj );


/** \fn     int ppm_imageprocessing_loader( configlist_t *nodelist , \
                                            configlist_t *distributedlist ,\
                                            configlist_t *inputlist );
```

```
 *  \brief   Invokes the image processing pipeline handling of the PPM.
 *
 *  \param   *nodelist          Name of the node(s)
 *  \param   *distributedlist   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
 */
int ppm_imageprocessing_loader( configlist_t *nodelist,
                                configlist_t *distributedlist,
                                configlist_t *inputlist);


/** \fn      int ppm_read_imageinput( configlist_t  *inputlist, \
                                      char          **sourcedir, \
                                      char          **targetdir);
 *  \brief   Reads the input file, which contains the needed information for
 *           image processing.
 *
 *  \param    *inputlist   List with name(s) of input file(s)
 *  \param   **sourcedir   Returns the source directory with images
 *  \param   **targetdir   Returns target directory
 *  \return                0 on success or -1 on any error
 */
int ppm_read_imageinput( configlist_t  *inputlist,
                         char          **sourcedir,
                         char          **targetdir);


/** \fn      int rmiximgprocclient_updateimagecounter( \
                                        rmix_remoteref_t *remoteobj,\
                                        unsigned int      imagecounter);
 *  \brief   Updates the image counter a the plug-in (client-side method stub).
 *
 *  \param   *remoteobj     ID of the remote object
 *  \param    imagecounter  New image counter value
 *  \return                 0 on success or -1 on any error
 */
int rmiximgprocclient_updateimagecounter( rmix_remoteref_t *remoteobj,
                                          unsigned int      imagecounter);


/** \fn      int rmiximgprocclient_updatepredecessor( \
                                        rmix_remoteref_t *remoteobj, \
                                        char             *successor, \
                                        unsigned int     *imagecounter);
 *  \brief   Updates the successor entry of a predecessor plug-in (client-side
 *           method stub).
 *
 *  \param   *remoteobj     ID of the remote object
 *  \param   *successor     New successor entry
 *  \param   *imagecounter  The returned image counter from the predecessor
 *  \return                 0 on success or -1 on any error
 */
int rmiximgprocclient_updatepredecessor( rmix_remoteref_t *remoteobj,
                                         char             *successor,
                                         unsigned int     *imagecounter);


/** \fn      int rmiximgprocclient_updatesuccessor( \
                                        rmix_remoteref_t *remoteobj, \
                                        char             *predecessor);
 *  \brief   Updates the predecessor entry of a successor plug-in (client-side
 *           method stub).
```

```
 *
 *  \param   *remoteobj      ID of the remote object
 *  \param   *predecessor    New predecessor entry
 *  \return                  0 on success or -1 on any error
 */
int rmiximgprocclient_updatesuccessor( rmix_remoteref_t *remoteobj,
                                       char             *predecessor);



/** \fn      RMIX_METHOD_CALL(rmixppm_repairpipe_call);
 *  \brief   Accepts calls for repairing the pipeline in case of an error.
 *           (server-side stub)
 *
 *  \param   object   The local object.
 *  \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
 *                    dynamically or explicitly set to NULL by this function. In
 *                    the case of variable arrays, the length value is allocated
 *                    before the call and is set to 0 if its variable array is
 *                    NULL.
 *  \param   outcnt   The output values count.
 *  \param   inary    The input values array. The values array and its containing
 *                    values are not modified or deallocated by this call.
 *  \param   incnt    The input values count.
 *  \return           0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmixppm_repairpipe_call);


/** \fn      int rmixppm_repairpipe (char *node);
 *  \brief   Tries to repair the pipeline in case of an error.
 *
 *  \param   *node  Node which cannot be accessed.
 *  \return         0 on success or -1 on any error.
 */
int rmixppm_repairpipe (char *node);


/** \fn      int rmiximgprocclient_sendworklist_oneway(
                                            rmix_remoteref_t *remoteobj);
 *  \brief   Triggers a plug-in unit to resend its backup list to its successor
 *           plug-in (client-side method stub).
 *
 *  \param     *remoteobj   ID of the remote object
 *  \return                 0 on success or -1 on any error
 */
int rmiximgprocclient_sendworklist_oneway (rmix_remoteref_t *remoteobj);


/*****************************************************************************
 *
 * Data
 *
 *****************************************************************************/

/** \var extern ppm_t ppm
 *  \brief Parallel Plug-in Manager "object".
 *
 *  An external "object" of the Parallel Plug-in Manager "class".
 */
extern ppm_t ppm;
```

```
#endif /* PPM_PPM_H */

/***************************************************************************
 *
 * END OF FILE
 *
 ***************************************************************************/
```

A.3.2.2. Source File for the PPM

```
/***************************************************************************
 *
 * Source file for the parallel plug-in manager module.
 * Copyright (c) Ronald Baumann.
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README     for general package information.
 * - INSTALL    for package install information.
 * - COPYING    for package license information and copying conditions.
 * - AUTHORS    for package authors information.
 * - ChangeLog for package changes information.
 *
 ***************************************************************************/

/** \file  ppm.c
 *  \brief Sorce file for the parallel plug-in manager module.
 *
 *  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
 *  of harness daemons.
 */


/***************************************************************************
 *
 * Includes
 *
 ***************************************************************************/

/* Main module header file. */
#include "ppm.h"


/***************************************************************************
 *
 * Data Types
 *
 ***************************************************************************/

/** \struct ppm_data_t
 *  \brief Parallel Plug-in Manager module data type.
 *
 *  Contains the mutex and the instances array.
 */
typedef struct
{
    pthread_mutex_t   mutex;              /* mutex                          */
```

```
    pthread_mutex_t  pipemutex;          /* pipe mutex                      */
    unsigned int     count;              /* instances array count           */
    struct
    {
        unsigned int handle;             /* instance handle                 */
    }*instances;                         /* instances array                 */
    unsigned int     handle_rmix;        /* Handle for rmix plug-in.        */
    rmix_localref_t *localref_harness;   /* exported Harness kernel         */
    rmix_localref_t *localref_ppm;       /* exported ppm plug-in            */
    configlist_t    *usednodes;          /* nodes used for distr. plug-in   */
    configlist_t    *freenodes;          /* nodes available for distr. plug-in */
    char            *ppmnode;            /* node name which runs the PPM    */
    configlist_t    *dist_plugins;       /* names of the distributed plug-ins */
    char            *sourcedir;          /* source directory with images    */
    char            *targetdir;          /* target directory for images     */
}ppm_data_t;


/***************************************************************************
 *
 * Function Prototypes
 *
 ***************************************************************************/

/** \fn     HARNESS_PLUGINS_INIT(ppm_init);
 *  \brief  Initializes the PPM plug-in.
 *
 *  \param  handle The plug-in instance handle.
 *  \return        0 on success or -1 on any error with errno set
 *                 appropriately.
 */
HARNESS_PLUGINS_INIT(ppm_init);


/** \fn     HARNESS_PLUGINS_FINI(ppm_fini);
 *  \brief  Finalizes the PPM plug-in.
 *
 *  \param  handle The plug-in instance handle.
 *  \return        0 on success or -1 on any error with errno set appropriately.
 */
HARNESS_PLUGINS_FINI(ppm_fini);


/***************************************************************************
 *
 * Data
 *
 ***************************************************************************/

/** \var   const rmix_method_t rmixppm_methods[RMIXPPM_METHODS_COUNT]
 *  \brief Server-side method descriptors for rmix ppm.
 */
const rmix_method_t rmixppm_methods[RMIXPPM_METHODS_COUNT] =
    RMIXPPM_METHODS;


/** \var   const rmix_interface_t rmixppm_interface
 *  \brief Server-side interface for rmix ppm.
 */
const rmix_interface_t rmixppm_interface =
{
    RMIXPPM_METHODS_COUNT,               /* method descriptor count */
    (rmix_method_t*)rmixppm_methods      /* method descriptor array */
```

```
};

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

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

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

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

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

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

```
 *  pointers.
 */
ppm_t ppm =
{
    {
        LIBPPM_VERSION_CURRENT,     /* current version         */
        LIBPPM_VERSION_REVISION,    /* current revision        */
        LIBPPM_VERSION_AGE,         /* version age             */
        LIBPPM_VERSION_FIRST        /* first supported version */
    }                               /* plug-in version         */
};


/******************************************************************************
 *
 * Functions
 *
 ******************************************************************************/

/*
 * Initializes the Integral-Loader plug-in.
 *
 * handle = The plug-in instance handle.
 * return = 0 on success or -1 on any error with errno set appropriately.
 */
#undef  __FUNC__
#define __FUNC__ "ppm_init"
HARNESS_PLUGINS_INIT(ppm_init)
{
    int             error;
    void           *instances;
    unsigned int    index;

    PPM_PRINT((
        PPM_INFO(libppm is starting)))

    /* Lock ppm plug-in mutex. */
    if (0 != (error = pthread_mutex_lock(&ppm_data.mutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to lock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }

    /* Search for handle in instances array. */
    for (index = 0; index < ppm_data.count; index++)
    {
        if (ppm_data.instances[index].handle == handle)
        {
            PPM_PRINT((
                PPM_WARN(handle is already in instances array)))
            harness_syserr();

            /* Unlock integral-loader plug-in mutex. */
            if (0 != (error =
                      pthread_mutex_unlock(&ppm_data.mutex)))
            {
                errno = error;
                PPM_PRINT((
                    PPM_WARN(unable to unlock ppm plug-in mutex)))
                harness_syserr();
```

```
            }
            return -1;
        }
    }

    /* Increase instances array. */
    index = ppm_data.count;
    ppm_data.count++;
    /* Reallocate instances array. */
    if (NULL == (instances = realloc(ppm_data.instances,
                                     ppm_data.count *
                              sizeof(ppm_data.instances[0]))))
    {
        PPM_PRINT((
            PPM_WARN(unable to reallocate instances array)))
        /* Unlock integral-loader plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&ppm_data.mutex)))
        {
            errno = error;
            PPM_PRINT((
                PPM_WARN(unable to unlock ppm plug-in mutex)))
            harness_syserr();
        }
        return -1;
    }

    ppm_data.instances = instances;
    /* Save instances array entry. */
    ppm_data.instances[index].handle = handle;
    /* Check for first initialization. */
    if (1 == ppm_data.count)
    {
        /**************************/
        /* PUT YOUR INIT CODE HERE */
        /**************************/
        ppm_init_rmix();
        ppm_start();
    }

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

    return 0;
}


/*
 * Finalizes the parallel plug-in manager.
 *
 * handle = The plug-in instance handle.
 * return = 0 on success or -1 on any error with errno set appropriately.
 */
#undef __FUNC__
#define __FUNC__ "ppm_fini"
HARNESS_PLUGINS_FINI(ppm_fini)
{
```

```
    int             error;
    void           *instances;
    unsigned int    index;

    PPM_PRINT((
        PPM_INFO(libppm is shutting down)))

    /* Lock integral-loader plug-in mutex. */
    if (0 != (error = pthread_mutex_lock(&ppm_data.mutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to lock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }

    /* Search for handle in instances array. */
    for (index = 0; index < ppm_data.count; index++)
    {
        if (ppm_data.instances[index].handle == handle)
        {
            break;
        }
    }

    /* Check if handle is not in instances array. */
    if (index == ppm_data.count)
    {
        PPM_PRINT((
            PPM_WARN(handle is not in instances array)))
        harness_syserr();
        /* Unlock integral-loader plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&ppm_data.mutex)))
        {
            errno = error;
            PPM_PRINT((
                PPM_WARN(unable to unlock ppm plug-in mutex)))
            harness_syserr();
        }
        return -1;
    }

    /* Remove instance from instances array. */
    ppm_data.count--;
    memmove(ppm_data.instances + index,
            ppm_data.instances + index +1,
            (ppm_data.count     - index) *
            sizeof(unsigned int));

    /* Reallocate instances array. */
    if (0 == ppm_data.count)
    {
        free(ppm_data.instances);
        ppm_data.instances = NULL;
    }
    else if (NULL == (instances = realloc(ppm_data.instances,
                                          ppm_data.count *
                                          sizeof(ppm_data.instances[0]))))
    {
        PPM_PRINT((
            PPM_WARN(unable to reallocate instances array)))
    }
```

```
    else
    {
        ppm_data.instances = instances;
    }
    /* Check for last finalization. */
    if (0 == ppm_data.count)
    {
        /**************************/
        /* PUT YOUR FINI CODE HERE */
        /**************************/
        ppm_fini_rmix();
        configlist_delete_list(&ppm_data.usednodes);
        configlist_delete_list(&ppm_data.freenodes);
        configlist_delete_list(&ppm_data.dist_plugins);
        FREE(ppm_data.sourcedir);
        FREE(ppm_data.targetdir);

    }

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


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

    PPM_PRINT((
        PPM_INFO(start initialization of RMIX)))

    /* Load RMIX plug-in and initialize RMIX */
    ret = harness_plugins_load( &ppm_data.handle_rmix, "libharness-rmix",
                                HARNESS_PLUGINS_EXPORT);
    if (ret != 0)
    {
        PPM_PRINT((
            PPM_WARN(unable to load RMIX plug-in)))
        return -1;
    }

    /* Exports Harness while forcing 1000 as specific object handle */
    ret = harness_rmix_export4( &ppm_data.localref_harness,
                               "PROTOCOL=RPC OBJECTID=1000");
    if (ret != 0)
    {
        PPM_PRINT((
            PPM_WARN(unable to export Harness kernel)))
```

```
            return −1;
        }

        /* Exports ppm plug−in while forcing 1003 as specific object handle */
        ret = rmix_export4( &ppm_data.localref_ppm,
                    "PROTOCOL=RPC OBJECTID=1003", NULL, &rmixppm_interface);
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to export ppm interface)))
            return −1;
        }

        PPM_PRINT((
            PPM_INFO(RMIX initialized)))

        return 0;
}


/*
 *  Finalizes rmix.
 *
 *  \return 0 on success or −1 on any error
 */
#undef   __FUNC__
#define __FUNC__ "ppm_fini_rmix"
int ppm_fini_rmix()
{
        int ret;
        int err = 0;

        PPM_PRINT((
            PPM_INFO(start finalization of RMIX)))

        /* Unexports ppm plug−in */
        ret = rmix_unexport( &ppm_data.localref_ppm);
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to unexport ppm interface)))
            err = −1;
        }

        /* Unexports Harness */
        ret = harness_rmix_unexport( &ppm_data.localref_harness);
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to unexport Harness kernel)))
            err = −1;
        }

        /* Unload RMIX plug−in */
        ret = harness_plugins_unload( ppm_data.handle_rmix);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(unable to unload RMIX plug−in)))
            err = −1;
        }

        PPM_PRINT((
            PPM_INFO(RMIX finalized)))

        return err;
```

```
}

/*
 *  Invokes the Parallel Plug-in Manager execution.
 *
 *  \return 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "ppm_start"
int ppm_start()
{
    int ret;
    int mode = 0;          /* 1 = replicated, 2 = distributed parallel plug-in */
    char *ppmnode = NULL; /* node of the parallel plug-in manager */

    configlist_t *p_nodelist;           /* linked list with node addresse(s) */
    configlist_t *p_replicatedlist;     /* linked list with plug-in name(s)  */
    configlist_t *p_distributedlist;    /* linked list with plug-in name(s)  */
    configlist_t *p_inputlist;          /* linked list with input file(s)    */

    p_nodelist         = NULL;
    p_replicatedlist   = NULL;
    p_distributedlist  = NULL;
    p_inputlist        = NULL;

    /* read configuration data */
    ret = configuration_read_file( &p_nodelist, &p_replicatedlist,
                                   &p_distributedlist, &p_inputlist, &mode,
                                   &ppmnode);
    if (ret == -1)
    {
        PPM_PRINT((
            PPM_WARN(could not read configuration file)))
        configlist_delete_list(&p_nodelist);
        configlist_delete_list(&p_replicatedlist);
        configlist_delete_list(&p_distributedlist);
        configlist_delete_list(&p_inputlist);
        FREE(ppmnode);
        return -1;
    }

    /* debug print of the lists */
    configlist_print_list(p_nodelist);
    configlist_print_list(p_replicatedlist);
    configlist_print_list(p_distributedlist);
    configlist_print_list(p_inputlist);
    PPM_PRINT((
        PPM_INFO(mode = %d), mode))
    PPM_PRINT((
        PPM_INFO(ppmnode = %s), ppmnode))

    /************************************************/
    /* PUT YOUR PARALLEL PLUG-IN MANAGER CODE HERE */
    /************************************************/

    /* load the Harness kernel with the parallel plug-in units */
    if (mode == 1)
        ret = ppm_load_harnesskernel_replicated( p_nodelist, p_replicatedlist);
    else if (mode == 2)
        ret = ppm_load_harnesskernel_distributed( &p_nodelist,
                                                  p_distributedlist,
                                                  ppmnode);
```

```
else
{
    PPM_PRINT((
        PPM_WARN(no appropriate parallel plug−in type set in conf file)))
    configlist_delete_list(&p_nodelist);
    configlist_delete_list(&p_replicatedlist);
    configlist_delete_list(&p_distributedlist);
    configlist_delete_list(&p_inputlist);
    FREE(ppmnode);
    return −1;
}
if ( ret == −1 )
{
    PPM_PRINT((
        PPM_WARN(could not load Harness kernel)))
    ppm_unload_harnesskernel_soft( p_nodelist);
    configlist_delete_list(&p_nodelist);
    configlist_delete_list(&p_replicatedlist);
    configlist_delete_list(&p_distributedlist);
    configlist_delete_list(&p_inputlist);
    FREE(ppmnode);
    return −1;
}

/***********************************************/
/* PUT YOUR PARALLEL PLUG−IN MANAGER CODE HERE */
/***********************************************/

/* start the specified plug−in handling for Monte Carlo or image
   processing */
if (mode == 1)
{
    ret = ppm_monte_carlo_loader( p_nodelist, p_replicatedlist,
                                  p_inputlist);
    if ( ret == −1 )
    {
        PPM_PRINT((
            PPM_WARN(could not execute parallel plug−in application)))
    }

    /* after executing the calculation the ppm is responsible for the
       shutdown of the involved plug−in units */
    ret = ppm_unload_harnesskernel_soft( p_nodelist);
    if ( ret == −1 )
    {
        PPM_PRINT((
            PPM_WARN(could not unload Harness kernel)))
         configlist_delete_list(&p_nodelist);
         configlist_delete_list(&p_replicatedlist);
         configlist_delete_list(&p_distributedlist);
         configlist_delete_list(&p_inputlist);
        FREE(ppmnode);
        return −1;
    }

}
else if (mode == 2)
{
    /* in case of the image processing pipeline, each plug−in shutdown
       itself when all images were processed */
    ret = ppm_imageprocessing_loader( p_nodelist, p_distributedlist,
                                      p_inputlist);
    if ( ret == −1 )
```

```
        {
            PPM_PRINT((
                PPM_WARN(could not execute parallel plug-in application)))
            /* unload the kernel */
            ppm_unload_harnesskernel_soft( p_nodelist);
            configlist_delete_list(&p_nodelist);
            configlist_delete_list(&p_replicatedlist);
            configlist_delete_list(&p_distributedlist);
            configlist_delete_list(&p_inputlist);
            configlist_delete_list(&ppm_data.usednodes);
            configlist_delete_list(&ppm_data.freenodes);
            configlist_delete_list(&ppm_data.dist_plugins);
            FREE(ppm_data.sourcedir);
            FREE(ppm_data.targetdir);
            FREE(ppmnode);
            return -1;
        }
    }
    else
    {
            PPM_PRINT((
                PPM_WARN(no appropriate parallel plug-in mode selected in
                         configuration file)))
            configlist_delete_list(&p_nodelist);
            configlist_delete_list(&p_replicatedlist);
            configlist_delete_list(&p_distributedlist);
            configlist_delete_list(&p_inputlist);
            FREE(ppmnode);
            return -1;
    }


    /**********************************************/
    /*                                            */
    /**********************************************/

    configlist_delete_list(&p_nodelist);
    configlist_delete_list(&p_replicatedlist);
    configlist_delete_list(&p_distributedlist);
    configlist_delete_list(&p_inputlist);
    FREE(ppmnode);
    return 0;
}


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

```
    PPM_PRINT((
        PPM_INFO(start loading harness kernel and parallel plug-in)))

    num_nodes   = configlist_listsize(nodelist);
    num_plugins = configlist_listsize(pluginlist);

    /* check the availability of nodes and plug-ins */
    if ((num_plugins == 0) || (num_nodes == 0))
    {
        PPM_PRINT((
            PPM_WARN(no nodes or plug-in names available)))
        return -1;
    }

    /* check if a filename was stored */
    if ( (configlist_check_entries(   nodelist) == -1) ||
         (configlist_check_entries( pluginlist) == -1))
    {
        PPM_PRINT((
            PPM_WARN(some node or plug-in entries have no name)))
        return -1;
    }

    /* replicated parallel plug-in */
    for( i=0; i<num_nodes; i++)
    {
#ifdef DEBUG
        /* start with debug output */
        command = (char*)malloc( sizeof(char) * (strlen(nodelist->name) +
                                 strlen(pluginlist->name) + 28) );
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "ssh %s harnessd.debug --load=%s",
                 nodelist->name, pluginlist->name);
#else
        /* start without debug output */
        command = (char*)malloc( sizeof(char) * (strlen(nodelist->name) +
                                 strlen(pluginlist->name) + 22) );
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "ssh %s harnessd --load=%s", nodelist->name,
                 pluginlist->name);
#endif
        /* execute the ssh command */
        if ( 0 != harness_processes_execute(&handle, command, NULL) )
        {
            PPM_PRINT((
                PPM_WARN(could not start Harness kernel %s),
                         nodelist->name))
        }

        FREE(command);
        nodelist = nodelist->next;
    }
```

```
    return 0;
}


/*
 *  Loads a distributed parallel plug-in on the Harness kernels specified
 *  in the conf file.
 *
 *  \param  **nodelist   Name of the node(s)
 *  \param   *pluginlist Name of the parallel plug-in
 *  \param   *ppmnode    Node running the PPM
 *  \return              0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "ppm_load_harnesskernel_distributed"
int ppm_load_harnesskernel_distributed( configlist_t **nodelist,
                                        configlist_t  *pluginlist,
                                        char          *ppmnode)
{
    int    handle;
    int    num_plugins;
    int    num_nodes;
    int    ret;
    char *command = NULL;
    rmix_remoteref_t *remoteobj;
    configlist_t      *tmpnodelist;
    configlist_t      *tmppluginlist;

    PPM_PRINT((
        PPM_INFO(start loading harness kernel and distributed plug-in)))

    tmpnodelist   = *nodelist;
    tmppluginlist =  pluginlist;

    num_nodes   = configlist_listsize(tmpnodelist);
    num_plugins = configlist_listsize(pluginlist);

    /* check the availability of nodes and plug-ins */
    if ((num_plugins == 0) || (num_nodes == 0))
    {
        PPM_PRINT((
            PPM_WARN(no nodes or plug-in names available)))
        return -1;
    }

    /* check if node and plug-in names were stored */
    if ( (configlist_check_entries(tmpnodelist) == -1) ||
         (configlist_check_entries( pluginlist) == -1))
    {
        PPM_PRINT((
            PPM_WARN(some node or plug-in entries have no name)))
        return -1;
    }


    /* check if there are not more plug-ins than nodes */
    if ( num_plugins > num_nodes )
    {
        PPM_PRINT((
            PPM_WARN(more plug-in units than available nodes)))
        return -1;
    }
```

```
    /* start loading the plug-ins */
    while(tmppluginlist != NULL)
    {
#ifdef DEBUG
        /* create the ssh start command with debug output */
        command = (char*)malloc( sizeof(char) * (strlen(tmpnodelist->name) +
                                 strlen(tmppluginlist->name) + 28) );
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "ssh %s harnessd.debug --load=%s",
                 tmpnodelist->name, tmppluginlist->name);
#else
        /* create the ssh start command without debug output */
        command = (char*)malloc( sizeof(char) * (strlen(tmpnodelist->name) +
                                 strlen(tmppluginlist->name) + 22) );
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "ssh %s harnessd --load=%s", tmpnodelist->name,
                 tmppluginlist->name);
#endif
        /* execute the ssh command */
        if ( 0 != harness_processes_execute(&handle, command, NULL) )
        {
            PPM_PRINT((
                PPM_WARN(could not start Harness kernel %s),
                         tmpnodelist->name))

            FREE(command);

            /* in case of an error delete the current node from the list */
            ret = configlist_delete_entry( configlist_find_elementposition(
                                          tmpnodelist->name, *nodelist),
                                          nodelist);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not delete node from nodelist)))
                return -1;
            }
            num_nodes--;

            /* check if there are still enough nodes */
            if (num_nodes < num_plugins)
            {
                PPM_PRINT((
                    PPM_WARN(more plug-in units than available nodes)))
                return -1;
            }
            else
            {
                /* try the next node */
                tmpnodelist = tmpnodelist->next;
                continue;
            }
        }
```

```
        FREE(command);

        /* check whether the kernel was loaded */

        /* set parameters for creating the remote references */
        command = (char*)malloc( sizeof(char) *
                        (strlen("PROTOCOL=RPC OBJECTID=1002 HOST=%s") +
                         strlen(tmpnodelist->name) - 1 ));
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "PROTOCOL=RPC OBJECTID=1002 HOST=%s",
                 tmpnodelist->name);

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

            FREE(command);

            /* delete current node from list */
            ret = configlist_delete_entry( configlist_find_elementposition(
                                           tmpnodelist->name, *nodelist),
                                           nodelist);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not delete node from nodelist)))
                return -1;
            }

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

        FREE(command);

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

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

```
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not contact plug-in on node %s),
                         tmpnodelist->name))

            ret = rmix_remoteref_destroy(&remoteobj);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }

            /* delete current node from the list */
            ret = configlist_delete_entry( configlist_find_elementposition(
                                           tmpnodelist->name, *nodelist),
                                           nodelist);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not delete node from nodelist)))
                return -1;
            }

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

        ret = rmix_remoteref_destroy(&remoteobj);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not destroy remote object references)))
        }

        /* load next plug-in on the next free node */
        tmpnodelist = tmpnodelist->next;
        tmppluginlist = tmppluginlist->next;
    }

    /* distinguish between standard loading of distributed plug-ins and repair a
       pipeline,
       if a distributed plug-in is loaded, the lists with used and free nodes
       have to be generated */
    if (ppmnode != NULL)
    {
        /* copy the nodes to the specific list (usednodes and freenodes) */
        tmpnodelist = *nodelist;
        tmppluginlist =  pluginlist;

        /* generate used node list */
```

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

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

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

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


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

    num_nodes = configlist_listsize(nodelist);
```

```
    /* terminate all kernel */
    for ( i=0; i<num_nodes; i++)
    {
#ifdef DEBUG
        /* kill harness daemon which used debug */
        command = (char*)malloc( sizeof(char) *
                                (strlen("ssh %s killall harnessd.debug") +
                                 strlen(nodelist->name) - 1 ));
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "ssh %s killall harnessd.debug", nodelist->name);
#else
        /* kill harness daemon which did not use debug */
        command = (char*)malloc( sizeof(char) *
                                (strlen("ssh %s killall harnessd") +
                                 strlen(nodelist->name) - 1 ));
        if (command == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
        }
        sprintf( command, "ssh %s killall harnessd", nodelist->name);
#endif
        /* perform ssh command */
        if ( 0 != harness_processes_execute(&handle, command, NULL) )
            PPM_PRINT((
                PPM_WARN(could not shutdown Harness kernel %s), nodelist->name))

        FREE(command);
        nodelist = nodelist->next;
    }

    return 0;
}


/*
 *  Unoads the Harness kernels on the nodes specified in the conf file.
 *  The kernels are completely killed inclusive all running plug-ins.
 *  For terminating the kernel the internal shutdown function is used.
 *
 *  \param   *nodelist    Name of the node(s)
 *  \return               0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "ppm_unload_harnesskernel_soft"
int ppm_unload_harnesskernel_soft( configlist_t *nodelist)
{
    rmix_remoteref_t *remoteobj;
    int i;
    int ret;
    int num_nodes;

    num_nodes = configlist_listsize(nodelist);

    /* terminate all kernel */
    for ( i=0; i<num_nodes; i++)
    {
```

```
        /* prepare remote reference */
        ret = ppm_createremoteref( &remoteobj, "1000", nodelist->name);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not create remote object references for %s),
                         nodelist->name))

            nodelist = nodelist->next;
            continue;
        }

        /* call the remote shutdown function */
        harnessclient_kernel_shutdown(remoteobj);

        ret = rmix_remoteref_destroy(&remoteobj);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not destroy remote object references)))
        }

        /* move to the next node */
        nodelist = nodelist->next;
    }

    return 0;
}


/*
 *  The process sleeps a preset time interval.
 *
 *  \param   sec_dlay   Delay in seconds
 *  \param  usec_dlay   Delay in micro seconds
 */
#undef  __FUNC__
#define __FUNC__ "ppm_sleep"
void ppm_sleep(int sec_dlay, int usec_dlay)
{
    struct timeval tv;
    if (sec_dlay > 0)
    {
        time_t start = time(0);
        /* in a loop to be signal-resilient */
        for (;;)
        {
            tv.tv_sec = sec_dlay - (time(0) - start);
            if (tv.tv_sec <= 0) break;
            tv.tv_usec = 0;
            (void)select(0, 0, 0, 0, &tv);
        }
    }
    /* don't worry about signals for usecs */
    if (usec_dlay > 0)
    {
        tv.tv_sec = 0;
        tv.tv_usec = usec_dlay;
        (void)select(0, 0, 0, 0, &tv);
    }

}
```

```

/*
 *  Creates a remote reference specified by objectid and node. The RPC protocol
 *  is used.
 *
 *  \param    **remoteobj  ID of the remote object
 *  \param     *objectid   ID of the exported object
 *  \param     *node       Name of the node
 *  \return                0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "ppm_createremoteref"
int ppm_createremoteref( rmix_remoteref_t **remoteobj,
                         char              *objectid,
                         char              *node)
{
    int   ret;
    char *remoteparameters;

    /* prepare parameter for creating remote reference */
    remoteparameters = (char*)malloc( sizeof(char) *
                          (strlen("PROTOCOL=RPC OBJECTID=%s HOST=%s") +
                           strlen(objectid) +
                           strlen(node) - 3 ));
    if (remoteparameters == NULL)
    {
        PPM_PRINT((
                PPM_WARN(could not allocate memory)))
            return -1;
    }
    sprintf( remoteparameters, "PROTOCOL=RPC OBJECTID=%s HOST=%s",
            objectid, node);

    /* create the remote reference */
    ret = rmix_remoteref_create6( &(*remoteobj), remoteparameters);
    if (ret != 0)
    {
        PPM_PRINT((
            PPM_WARN(could not create remote object references)))
        return -1;
    }

    FREE(remoteparameters);
    return 0;
}


/*****************************************************************************
 *
 * Monte Carlo section
 *
 *****************************************************************************/

/*
 *  Invokes the Monte Carlo parallel plug-in execution.
 *
 *  \param  *nodelist         Name of the input file
 *  \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
 */
#undef  __FUNC__
#define __FUNC__ "ppm_monte_carlo_loader"
```

```
int ppm_monte_carlo_loader( configlist_t *nodelist,
                            configlist_t *replicatedlist,
                            configlist_t *inputlist)
{
    int ret;

    int          num_of_coeffs;   /* number of coefficients            */
    unsigned int iterations;      /* amount of random numbers          */
    double      *coeffs;          /* array for storing the coefficients */
    double       lower_border;    /* lower border for the calculation area */
    double       upper_border;    /* upper border of the calculation area  */

    /* read input data */
    ret = ppm_read_integralinput( inputlist, &num_of_coeffs, &coeffs,
                                  &iterations, &lower_border, &upper_border);
    if ( ret == -1 )
    {
        PPM_PRINT((
            PPM_WARN(could not read input file)))
        return -1;
    }

    /* schedule the integral intervals and call the remote calculation
       functions */
    ret = ppm_schedule_integral( nodelist, replicatedlist, num_of_coeffs,
                                 coeffs, iterations, lower_border,
                                 upper_border);
    if ( ret == -1 )
    {
        PPM_PRINT((
            PPM_WARN(could not schedule the integral data)))
        FREE(coeffs);
        return -1;
    }

    FREE(coeffs);

    return 0;
}


/*
 * Invokes rmixintegral_integration (client-side method stub).
 * Sends the integration parameters to the server and starts the integrstion
 * process.
 *
 * \param  **invokeref     The invocation reference (also return).
 * \param  *remoteobject   The remote object reference.
 * \param   iterations     The number of the generated random numbers.
 * \param   lowerbound     The lower boundary of the integration interval.
 * \param   upperbound     The upper boundary of the integration interval.
 * \param   numcoeffs      The number of function coefficients.
 * \param  *coeffs         The coefficients.
 * \return                 0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmixintegralclient_integration_send"
int rmixintegralclient_integration_send( rmix_invokeref_t **invokeref,
                                         rmix_remoteref_t  *remoteobject,
                                         unsigned int       iterations,
                                         double             lowerbound,
                                         double             upperbound,
                                         unsigned int       numcoeffs,
```

```
                                             double              *coeffs)
{
    const void *inary[5];

    /* Prepare parameters. */
    inary[0] = &iterations;
    inary[1] = &lowerbound;
    inary[2] = &upperbound;
    inary[3] = &numcoeffs;
    inary[4] =  coeffs;

    /* Invoke remote object method by using an asynchronous call. */
    if (0 != rmix_send(invokeref, remoteobject, &rmixintegralclient_interface,
                       RMIXINTEGRAL_METHODS_INTEGRATION_INDEX, inary, 5))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}


/*
 * Retrieves rmixintegral_integration invocation result (client-side method
 * stub).
 * Returns the output of a previous asynchonous invocation that sends the
 * integration parameters to the server and starts the integration
 * process.
 *
 * \param  **invokeref   The invocation reference (also return).
 * \param  *integral     The result of the integration.
 * \return               0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmixintegralclient_integration_retrieve"
int rmixintegralclient_integration_retrieve( rmix_invokeref_t **invokeref,
                                             double             *integral)
{
    int   result;
    void *outary[2];
#ifdef DEBUG
    /* Check invokeref parameter. */
    if (NULL == invokeref)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(invokeref parameter is null)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare output. */
    outary[0] = &result;
    outary[1] =  integral;

    /* Retrieve invocation result. */
    if (0 != rmix_retrieve(invokeref, outary, 2))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
```

```
        errno = errno2;
        return -1;
    }
    return result;
}



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

    PPM_PRINT((
        PPM_INFO(start reading input file)))

    /* size of the input file list */
    if (configlist_listsize(inputlist) == 0)
    {
        PPM_PRINT((
            PPM_WARN(no input file in input list)))
        return -1;
    }

    /* only the first mentioned file will be read in */
    if (configlist_listsize(inputlist) != 1)
    {
        PPM_PRINT((
            PPM_INFO(more than one input file entries found - try
                     using first one)))
    }

    /* check if a filename was stored */
    if (configlist_check_entries( inputlist) == -1)
    {
        PPM_PRINT((
            PPM_WARN(input file entry has no name)))
        return -1;
    }

    /* open the input file */
    dat_file = fopen(inputlist->name, "r");
    if (dat_file == NULL)
```

```
{
    PPM_PRINT((
        PPM_WARN(unable to open input file %s), inputlist->name))
    return -1;
}
else
{
    PPM_PRINT((
        PPM_INFO(input file is opened)))
}

/* read in the number of iterations */
ret = fscanf(dat_file, "%d", iterations);
if (ret == 0 || ret == -1)
{
    PPM_PRINT((
        PPM_WARN(unable to read iterations from file)))
    fclose(dat_file);
    return -1;
}

/* read in the boundaries */
ret = fscanf(dat_file, "%lf", lower);
if (ret == 0 || ret == -1)
{
    PPM_PRINT((
        PPM_WARN(unable to read lower boundary from file)))
    fclose(dat_file);
    return -1;
}

ret = fscanf(dat_file, "%lf", upper);
if (ret == 0 || ret == -1)
{
    PPM_PRINT((
        PPM_WARN(unable to read upper boundary from file)))
    fclose(dat_file);
    return -1;
}

/* read in the number of coefficients */
ret = fscanf(dat_file, "%d", number);
if (ret == 0 || ret == -1)
{
    PPM_PRINT((
        PPM_WARN(unable to read number of coefficients from
                                file)))
    fclose(dat_file);
    return -1;
}

/* create the array */
(*coeffs) = (double *)malloc( sizeof(double) * (*number));
if ((*coeffs) == NULL)
{
    PPM_PRINT((
        PPM_WARN(unable to allocate memory for coefficients)))
    fclose(dat_file);
    return -1;
}

/* read in the coefficients */
for (i=0; i<(*number); i++)
```

```
        {
            ret = fscanf(dat_file, "%lf", &(*coeffs)[i]);
            if (ret == 0 || ret == -1)
            {
                PPM_PRINT((
                    PPM_WARN(unable to read coefficients from file)))
                FREE(coeffs);
                fclose(dat_file);
                return -1;
            }
        }

    fclose(dat_file);
    return 0;
}


/*
 *  Partitions the integration interval and invokes the integral plug-in units.
 *  In case of a failure, a redistribution of missed interval parts is tried.
 *
 *  \param  *nodelist      Name of the input file
 *  \param  *pluginlist    Name of the parallel plug-in
 *  \param   numcoeffs     Number of coefficients
 *  \param  *coeffs        Array of coefficients
 *  \param   iterations    Amount of random numbers, which are used
 *  \param   lower_border  Lower boundary of the interval
 *  \param   upper_border  Upper boundary of the interval
 *  \return                0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "ppm_schedule_integral"
int ppm_schedule_integral( configlist_t *nodelist,
                           configlist_t *pluginlist,
                           unsigned int  numcoeffs,
                           double       *coeffs,
                           int           iterations,
                           double        lower_border,
                           double        upper_border)
{
    int      nodes;             /* number of nodes mentioned in conf file      */
    int      workingnodes;      /* number of currently working nodes           */
    double   interval;          /* interval for one plug-in unit               */
    double   lower;             /* lower boundary of temporary calculations    */
    double   upper;             /* upper boundary of temporary calculations    */
    double   result = 0;        /* result of the integration calculation       */
    double   inter  = 0;        /* intermediate result of one plug-in          */
    int      count;             /* counting variable                           */
    int      i, n;              /* variables used for loops                    */
    int      ret;               /* stores return values of functions           */
    int      position;          /* variable used to mark positions in arrays   */
    int      offset;            /* variable used to mark positions in arrays   */
    char    *remoteparameters;  /* used for creating remote reference objects  */

    unsigned int      errors;                /* counting occurring errors     */
    unsigned int     *errorpositions;        /* array storing the error nodes */
    rmix_remoteref_t **integral_tmpremotes;  /* temporary remote references   */
    rmix_remoteref_t **integral_remoterefs;  /* references after filtering    */
    rmix_invokeref_t **integral_invokerefs;  /* remote invokation references  */

    remoteparameters    = NULL;
    integral_tmpremotes = NULL;
    integral_remoterefs = NULL;
```

```
    integral_invokerefs = NULL;

    PPM_PRINT((
        PPM_INFO(start scheduling)))

    /* size of the node list */
    if ( (nodes = configlist_listsize(nodelist)) == 0)
    {
        PPM_PRINT((
            PPM_WARN(no nodes available)))
        return -1;
    }

    /* check if node names were stored */
    if (configlist_check_entries( nodelist) == -1)
    {
        PPM_PRINT((
            PPM_WARN(one or more node entries have no node name)))
        return -1;
    }

    /* size of plugin name list */
    if ( configlist_listsize(pluginlist) == 0)
    {
        PPM_PRINT((
            PPM_WARN(no plugin available)))
        return -1;
    }

    /* only the first mentioned plug-in will be used */
    if (configlist_listsize(pluginlist) != 1)
    {
        PPM_PRINT((
            PPM_INFO(more than one plug-in entry found - try using first one)))
    }

    /* check if a plugin name was stored */
    if (configlist_check_entries( pluginlist) == -1)
    {
        PPM_PRINT((
            PPM_WARN(plug-in entry has no stored name)))
        return -1;
    }

    /* allocate memory for the temporary remote references */
    integral_tmpremotes = (rmix_remoteref_t**) malloc(sizeof(rmix_remoteref_t*)
                                                      * nodes);
    if (integral_tmpremotes == NULL)
    {
        PPM_PRINT((
            PPM_WARN(could not allocate memory for remote references)))
        return -1;
    }

    /* workingnodes will store the number of currently available nodes */
    workingnodes = nodes;

    /* create remote references for all nodes mentioned in the conf file */
    for ( i=0; i<nodes; i++)
    {
        /* set parameters for creating the remote references */
        remoteparameters = (char*)malloc( sizeof(char) *
                           (strlen("PROTOCOL=RPC OBJECTID=1001 HOST=%s") +
```

```
                            strlen(nodelist->name) - 1 ));
        if (remoteparameters == NULL)
        {
            PPM_PRINT((
                PPM_WARN(could not allocate memory for remote parameters)))

            /* destroy already created remote references */
            for ( n=0; n<nodes; n++)
            {
                ret = rmix_remoteref_destroy( &integral_tmpremotes[i]);
                if (ret != 0)
                {
                    PPM_PRINT((
                        PPM_WARN(could not destroy remote object references)))
                }
            }

            /* free allocated memory */
            FREE(integral_tmpremotes);

            return -1;
        }
        sprintf( remoteparameters, "PROTOCOL=RPC OBJECTID=1001 HOST=%s",
                 nodelist->name);

        /* create references */
        ret = rmix_remoteref_create6( &integral_tmpremotes[i],
                                      remoteparameters);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not create remote object references)))

            /* if the remote reference cannot be created decrease the number
               of currently available nodes */
            workingnodes--;
            /* tag the node as not working */
            integral_tmpremotes[i] = NULL;
        }

        FREE(remoteparameters);

        /* create remote reference for the next node */
        nodelist = nodelist->next;
    }

    /* allocate memory for the remote references */
    integral_remoterefs = (rmix_remoteref_t**) malloc(sizeof(rmix_remoteref_t*)
                                                      * workingnodes);
    if (integral_remoterefs == NULL)
    {
        PPM_PRINT((
            PPM_WARN(could not allocate memory for remote references)))

        /* destroy already created temporary remote references */
        for ( i=0; i<nodes; i++)
        {
            ret = rmix_remoteref_destroy( &integral_tmpremotes[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }
```

```
        }
        FREE(integral_tmpremotes);
        return -1;
    }

    /* allocate memory for the invoke references */
    integral_invokerefs = (rmix_invokeref_t**) malloc(sizeof(rmix_invokeref_t*)
                                                  * workingnodes);
    if (integral_invokerefs == NULL)
    {
        PPM_PRINT((
            PPM_WARN(could not allocate memory for invoke references)))

        /* destroy already created temporary remote references */
        for ( i=0; i<nodes; i++)
        {
            ret = rmix_remoteref_destroy( &integral_tmpremotes[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }
        }
        FREE(integral_tmpremotes);
        FREE(integral_remoterefs);
        return -1;
    }


    /* identifies the current posittion in the remote references array for
       storing the next currently available node reference */
    count = 0;

    /* copy the working references */
    for ( i=0; i<nodes; i++)
    {
        if (integral_tmpremotes[i] != NULL)
        {
            /* copy reference if it isn't NULL */
            ret = rmix_remoteref_duplicate( &integral_remoterefs[count],
                                            integral_tmpremotes[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not copy remote object references)))

                /* destroy already created temporary remote references */
                for ( n=0; n<nodes; n++)
                {
                    ret = rmix_remoteref_destroy( &integral_tmpremotes[n]);
                    if (ret != 0)
                    {
                        PPM_PRINT((
                            PPM_WARN(could not destroy remote object
                                     references)))
                    }
                }

                /* destroy already copied working remote references */
                for ( n=0; n<workingnodes; n++)
                {
                    ret = rmix_remoteref_destroy( &integral_remoterefs[n]);
                    if (ret != 0)
```

```
                    {
                        PPM_PRINT((
                            PPM_WARN(could not destroy remote object
                                  references)))
                    }
                }
                FREE(integral_tmpremotes);
                FREE(integral_remoterefs);
                FREE(integral_invokerefs);
                return -1;
            }

            /* destroy copied reference */
            ret = rmix_remoteref_destroy( &integral_tmpremotes[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }
            count++;
        }
    }
    FREE(integral_tmpremotes);


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

    /* prepare the calculation data for distribution */

    /* interval size of one plug-in unit */
    interval = (upper_border - lower_border) / workingnodes;

    /* lower boundary for the first plug-in unit */
    lower = lower_border;

    /* upper boundary for the first plug-in unit */
    upper = lower + interval;

    /* counting occuring errors */
    errors = 0;

    /* call remote functions on the working nodes */
    for ( i=0; i<workingnodes; i++)
    {
        /* send data to the plug-ins asynchronously */
        if (integral_remoterefs[i] != NULL)
        {
            ret = rmixintegralclient_integration_send( &integral_invokerefs[i],
                                                        integral_remoterefs[i],
                                                        iterations,
                                                        lower,
                                                        upper,
                                                        numcoeffs,
                                                        coeffs);
            if (ret != 0)
            {
                /* if sending failed increase error counter */
                errors++;
                /* and tag the node as not available anymore */
                rmix_remoteref_destroy( &integral_remoterefs[i]);
```

```
                /* by setting the reference NULL */
                integral_remoterefs[i] = NULL;
            }

            /* borders for the next plug-in unit */
            lower = upper;
            upper = lower + interval;
        }
        else
            integral_invokerefs[i] = NULL;
    }

    /* call remote functions for receiving the results */
    for ( i = 0; i < workingnodes; i++)
    {
        /* only ask working nodes for results */
        if (integral_remoterefs[i] != NULL)
        {
            /* retrieve the result */
            ret = rmixintegralclient_integration_retrieve(
                                        &integral_invokerefs[i], &inter) ;
            if (ret != 0)
            {
                /* if retrieving fails increase error counter */
                errors++;

                /* and tag the node as not available anymore */
                rmix_remoteref_destroy( &integral_remoterefs[i]);
                /* by setting the reference NULL */
                integral_remoterefs[i] = NULL;
            }
            else
            {
                /* otherwise sum the result */
                result += inter;
            }
        }
    }

    /* if all nodes failed, there is no redistribution of work possible */
    if (errors == workingnodes)
    {
        PPM_PRINT((
            PPM_WARN(terminating program - all communication tries failed)))

        /* destroy remote references */
        for ( i=0; i<workingnodes; i++)
        {
            ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }
        }
        FREE(integral_remoterefs);
        FREE(integral_invokerefs);
        return -1;
    }


    count = 0;
    /* if errors occured start the redistribution of the failed calculations */
```

```
if (errors != 0)
{
    PPM_PRINT((
        PPM_INFO(starting error recovery)))

    /* the failed nodes will be stored in an array and with the help of
       the node number the missing calculation part will be recovered */
    errorpositions = (unsigned int*)malloc( sizeof(unsigned int) * errors);
    if (errorpositions == NULL)
    {
        PPM_PRINT((
            PPM_WARN(could not allocate memory for error positions)))

        /* destroy remote references */
        for ( i=0; i<workingnodes; i++)
        {
            ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }
        }
        FREE(integral_remoterefs);
        FREE(integral_invokerefs);
        return -1;
    }

    /* search for the failed nodes and store the rank number, the position
       in the nodelist identifies the chunk of data each node received */
    for ( i=0; i<workingnodes; i++)
    {
        if ( integral_remoterefs[i] == NULL)
        {
            errorpositions[count] = i;
        }
    }

    /* nodes are the currently available nodes, if nodes fail during the
       redistribution process nodes is decreased */
    nodes = workingnodes;

    /* count determines the failed calculation parts, if a calculation part
       is solved by redistribution count is decreased */
    count = errors;

    /* while still calculation parts are missing redistribute */
    while(count != 0)
    {
        /* if all nodes failed the redistribution isn't possible anymore */
        if (nodes == 0)
        {
            PPM_PRINT((
                PPM_WARN(all nodes failed during redistribution process)))

            /* destroy remote references */
            for ( i=0; i<workingnodes; i++)
            {
                ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
                if (ret != 0)
                {
                    PPM_PRINT((
                        PPM_WARN(could not destroy remote object
```

```
                        references)))
            }
        }
        FREE(integral_remoterefs);
        FREE(integral_invokerefs);
        FREE(errorpositions);
        return -1;
    }

    /* for loadbalancing position stores the node which got a new chunck
       for calculation as the last one */
    position = 0;

    /* solve all missed parts of the failed nodes */
    for ( i=0; i<errors; i++)
    {
        /* -1 tags the failed chunck as calculated */
        if (errorpositions[i] != -1)
        {
            /* calculate the lower boundary of the missed chunck */
            lower = lower_border + errorpositions[i] * (upper_border -
                    lower_border) / workingnodes;

            /* calculate the upper boundary of the missed chunck */
            upper = lower + (upper_border - lower_border)
                    / workingnodes;

            /* send the chunck to the next free and available node */
            for ( n=position; n<workingnodes; n++)
            {
                /* find an available node */
                if(integral_remoterefs[n] != NULL)
                {
                    /* send dataset */
                    ret = rmixintegralclient_integration_send(
                                        &integral_invokerefs[n],
                                         integral_remoterefs[n],
                                         iterations,
                                         lower,
                                         upper,
                                         numcoeffs,
                                         coeffs);
                    if (ret != 0)
                    {
                        /* if the sending fails tag the node as not */
                        ret = rmix_remoteref_destroy(
                                        &integral_remoterefs[n]);
                        if (ret != 0)
                        {
                            PPM_PRINT((
                                PPM_WARN(could not destroy remote object
                                    references)))
                        }
                        /* available anymore */
                        integral_remoterefs[n] = NULL;

                        /* decrease the number of currently available
                           nodes */
                        nodes--;

                        /* try to send the data chunck to the next
                           available node */
                        continue;
```

```
                }
                else
                {
                    /* store the position of the node which got the
                       chunck */
                    position = n;

                    /* calculate the next missing chunck and
                       redistribute it */
                    break;
                }
            }
        }
    }
}

position = 0;
offset   = 0;

/* collect the results */
for ( i=0; i<workingnodes; i++)
{
    /* from the still reachable nodes */
    if (integral_invokerefs[i] != NULL)
    {
        /* retrieve result */
        ret = rmixintegralclient_integration_retrieve(
                            &integral_invokerefs[i], &inter) ;
        if (ret != 0)
        {
            /* if the retrieving failed tag the node as failed */
            ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object
                             references)))
            }
            integral_remoterefs[i] = NULL;

            /* decrease the number of available nodes */
            nodes--;

            /* the algorithm assumes that all requests come back
               in the order as they were sent (depending on the
               position of their rank in the error array)
               if a part was not solved, the current position in
               the array of failed nodes cannot be set as solved, so
               an offset is increased */
            offset++;
        }
        else
        {
            /* sum up the result */
            result += inter;

            /* one missing calculation part is solved */
            count--;

            /* the current position in the array of failed nodes
               will be set as solved */
            for ( n=0; n<errors; n++)
            {
```

```
                            /* find the next unsolved chunck */
                            if(errorpositions[n] != -1)
                            {
                                /* if a chunck could not be received, the offset
                                   was set, the failed chunks will be overjumped
                                   and the next correct received chunk is reset
                                */
                                if (offset != 0)
                                {
                                    /* store the offset */
                                    if (position == 0)
                                        position = offset;
                                    /* decrease offset and */
                                    offset--;
                                    /* find the next unsolved chunk */
                                    continue;
                                }
                                else
                                {
                                    /* tag the chunck as solved */
                                    errorpositions[n] = -1;
                                    /* restore the offset */
                                    offset = position;
                                    position = 0;

                                    /* leave loop to get next result */
                                    break;
                                }
                            }
                        }
                    }
                }
            }
        }
        FREE(errorpositions);
    }

    PPM_PRINT((
        PPM_INFO(++++++++++++++++++++++++++++++++++++)))
    PPM_PRINT((
        PPM_INFO(integral = %lf), result))
    PPM_PRINT((
        PPM_INFO(++++++++++++++++++++++++++++++++++++)))

    /* destroy remote references */
    for ( i=0; i<workingnodes; i++)
    {
        ret = rmix_remoteref_destroy( &integral_remoterefs[i]);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not destroy remote object references)))
        }
    }

    FREE(integral_remoterefs);
    FREE(integral_invokerefs);
    return 0;
}

/**************************************************************************
 *
```

```
 * Image Processing section
 *
 ******************************************************************************/

/*
 *  Invokes the image processing pipeline.
 *
 *  \param  *nodelist           Name of the node(s)
 *  \param  *distributedlist    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
 */
#undef  __FUNC__
#define __FUNC__ "ppm_imageprocessing_loader"
int ppm_imageprocessing_loader( configlist_t *nodelist,
                                configlist_t *distributedlist,
                                configlist_t *inputlist)
{
    int ret;
    int i;
    unsigned int nodes;
    unsigned int plugins;

    rmix_remoteref_t **imgproc_remoterefs = NULL;

    char *p_sourcedir;       /* src directory with image files    */
    char *p_targetdir;       /* target directory for storing img  */

    configlist_t *tmp_pointer;

    p_sourcedir = NULL;
    p_targetdir = NULL;
    tmp_pointer = NULL;

    /* read input data */
    ret = ppm_read_imageinput( inputlist, &p_sourcedir, &p_targetdir);
    if ( ret == -1 )
    {
        PPM_PRINT((
            PPM_WARN(could not read input file)))
        FREE(p_sourcedir);
        FREE(p_targetdir);
        return -1;
    }

    /* print debug output */
    PPM_PRINT((
        PPM_INFO(source dir %s), p_sourcedir))
    PPM_PRINT((
        PPM_INFO(target dir %s), p_targetdir))

    /* copy the source and the target directories into global variable, for a
    possible pipeline restoration */
    ppm_data.sourcedir = (char*)malloc(sizeof(char) * (strlen(p_sourcedir) +1));
    if (ppm_data.sourcedir == NULL)
    {
        PPM_PRINT((
            PPM_WARN(could not allocate memory input file)))
        FREE(p_sourcedir);
        FREE(p_targetdir);
        return -1;
    }
    strcpy( ppm_data.sourcedir, p_sourcedir);
```

```
ppm_data.targetdir = (char*)malloc(sizeof(char) * (strlen(p_targetdir) +1));
if (ppm_data.targetdir == NULL)
{
    PPM_PRINT((
        PPM_WARN(could not allocate memory input file)))
    FREE(p_sourcedir);
    FREE(p_targetdir);
    return -1;
}
strcpy( ppm_data.targetdir, p_targetdir);

/* size of the node list */
if ( (nodes = configlist_listsize(nodelist)) == 0)
{
    PPM_PRINT((
        PPM_WARN(no nodes available)))
    return -1;
}

/* check if node names were stored */
if (configlist_check_entries( nodelist) == -1)
{
    PPM_PRINT((
        PPM_WARN(node entry has no node name)))
    return -1;
}

/* size of plugin name list */
if ( (plugins = configlist_listsize(distributedlist)) == 0)
{
    PPM_PRINT((
        PPM_WARN(no plugin available)))
    return -1;
}

/* allocate memory for the remote references */
imgproc_remoterefs = (rmix_remoteref_t**) malloc(sizeof(rmix_remoteref_t*)
                                               * plugins);
if (imgproc_remoterefs == NULL)
{
    PPM_PRINT((
        PPM_WARN(could not allocate memory for remote references)))
    return -1;
}

/* create remote references for all used nodes */
tmp_pointer = nodelist;
for ( i=0; i<plugins; i++)
{
    ret = ppm_createremoteref( &imgproc_remoterefs[i], "1002",
                               tmp_pointer->name);
    if (ret != 0)
    {
        PPM_PRINT((
            PPM_WARN(could not create remote object)))
        return -1;
    }
    tmp_pointer = tmp_pointer->next;
}

/* if the server isn't ready yet and cannot listen to a socket, an error
   occurs, the sleep should provide a certain amount of time for the just
   loaded plug-in to register the socket */
```

```
    ppm_sleep(2,0);


    tmp_pointer = nodelist;

    /* initialize the distributed parallel plug-in */
    /* "nada" identifies variables, which are not necessary for a particular
       pipeline unit, i.e. if a plug-in has as the name for the predecessor
       "nada", it knows that it has no predecessor */
    for ( i=0; i<plugins; i++)
    {
        /* the first plug-in unit has to open the images */
        if(i==0)
        {
            /* check if the "pipeline" only consists of one plug-in */
            if (plugins != 1)
                ret = rmiximgprocclient_initpipeline( imgproc_remoterefs[i], i,
                                                      p_sourcedir, "nada",
                                                      tmp_pointer->next->name,
                                                      "nada", ppm_data.ppmnode);
            else
                ret = rmiximgprocclient_initpipeline( imgproc_remoterefs[i], i,
                                                      p_sourcedir, p_targetdir,
                                                      "nada", "nada",
                                                      ppm_data.ppmnode);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))

                /* destroy remote and invoke references */
                for ( i=0; i<plugins; i++)
                {
                    ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
                    if (ret != 0)
                    {
                        PPM_PRINT((
                          PPM_WARN(could not destroy remote object references)))
                    }
                }
                FREE(imgproc_remoterefs);

                FREE(p_sourcedir);
                FREE(p_targetdir);

                return -1;
            }
        }
        /* the last plug-in unit has to store the images */
        else if(i==plugins-1)
        {
            ret = rmiximgprocclient_initpipeline( imgproc_remoterefs[i], i,
                                                  "nada", p_targetdir, "nada",
                                                  tmp_pointer->prev->name,
                                                  ppm_data.ppmnode);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))

                /* destroy remote and invoke references */
                for ( i=0; i<plugins; i++)
                {
```

```
                    ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
                    if (ret != 0)
                    {
                        PPM_PRINT((
                          PPM_WARN(could not destroy remote object references)))
                    }
                }
                FREE(imgproc_remoterefs);

                FREE(p_sourcedir);
                FREE(p_targetdir);

                return -1;
            }
        }
        /* every plug-in unit in between has to forward the images and
           acknowledgments */
        else
        {
            ret = rmiximgprocclient_initpipeline( imgproc_remoterefs[i], i,
                                                  "nada", "nada",
                                                  tmp_pointer->next->name,
                                                  tmp_pointer->prev->name,
                                                  ppm_data.ppmnode);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))

                /* destroy remote and invoke references */
                for ( i=0; i<plugins; i++)
                {
                    ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
                    if (ret != 0)
                    {
                        PPM_PRINT((
                          PPM_WARN(could not destroy remote object references)))
                    }
                }
                FREE(imgproc_remoterefs);

                FREE(p_sourcedir);
                FREE(p_targetdir);

                return -1;
            }
        }
        tmp_pointer = tmp_pointer->next;
    }

    /* invoke the first plug-in of the pipeline */
    ret = rmiximgprocclient_invokepipeline_oneway (imgproc_remoterefs[0]);
    if (ret != 0)
    {
        PPM_PRINT((
            PPM_WARN(could not invoke the pipeline)))

        /* destroy remote and invoke references */
        for ( i=0; i<plugins; i++)
        {
            ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
            if (ret != 0)
            {
```

```
                PPM_PRINT((
                    PPM_WARN(could not destroy remote object references)))
            }
        }
        FREE(imgproc_remoterefs);

        FREE(p_sourcedir);
        FREE(p_targetdir);

        return -1;
    }

    /* destroy remote and invoke references */
    for ( i=0; i<plugins; i++)
    {
        ret = rmix_remoteref_destroy( &imgproc_remoterefs[i]);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not destroy remote object references)))
        }
    }
    FREE(imgproc_remoterefs);

    FREE(p_sourcedir);
    FREE(p_targetdir);

    return 0;
}


/*
 * Reads the input file, which contains the needed information for image
 * processing.
 *
 * \param   *inputlist    List with name(s) of input file(s)
 * \param  **sourcedir    Returns list with directories of images
 * \param  **targetdir    Returns target directory
 * \return                0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "ppm_read_imageinput"
int ppm_read_imageinput( configlist_t  *inputlist,
                         char          **sourcedir,
                         char          **targetdir)
{
    /* build the input file structure */
    cfg_opt_t opts[] =
    {
        CFG_STR( SOURCEDIR, NULL, CFGF_NONE),
        CFG_STR( TARGETDIR, NULL, CFGF_NONE),
        CFG_END()
    };

    cfg_t *cfg;

    PPM_PRINT((
        PPM_INFO(start reading imgproc input file)))

    /* size of the input file list */
    if (configlist_listsize(inputlist) == 0)
    {
        PPM_PRINT((
```

```
            PPM_WARN(no input file in input list)))
        return -1;
    }

    /* only the second mentioned file will be read in */
    if (configlist_listsize(inputlist) != 1)
    {
        PPM_PRINT((
            PPM_INFO(more than one input file entries found - try
                     using second one)))
    }

    /* check if a filename was stored */
    if (configlist_check_entries( inputlist) == -1)
    {
        PPM_PRINT((
            PPM_WARN(input file entry has no name)))
        return -1;
    }


    /* initialize the conf file structure */
    cfg = cfg_init(opts, CFGF_NONE);

    /* read the input file */
    if(cfg_parse(cfg, inputlist->next->name) == CFG_PARSE_ERROR)
        return -1;

    /* extract target directory */
    if (cfg_getstr(cfg, TARGETDIR) == NULL)
    {
        (*targetdir) = NULL;
        PPM_PRINT((
            PPM_WARN(no target directory entry in input file)))
        return -1;
    }

    /* allocate memory for the target directory */
    (*targetdir) = (char*)malloc(sizeof(char)*(strlen(cfg_getstr(cfg,
                                   TARGETDIR))+1));
    if ( (*targetdir) == NULL )
    {
        PPM_PRINT((
            PPM_WARN(allocating of memory failed)))
        return -1;
    }

    /* store the information */
    strcpy( (*targetdir), cfg_getstr(cfg, TARGETDIR));

    /* extract source directory */
    if (cfg_getstr( cfg, SOURCEDIR) == NULL)
    {
        (*sourcedir) = NULL;
        PPM_PRINT((
            PPM_WARN(no source directory entry in input file)))
        return -1;
    }
    else
    {
        (*sourcedir) = (char*)malloc( sizeof(char) *
                                      (strlen(cfg_getstr(cfg, SOURCEDIR))+1));
        if ( (*sourcedir) == NULL )
```

```
        {
            PPM_PRINT((
                PPM_WARN(allocating of memory failed)))
            return -1;
        }

        /* store the information */
        strcpy( (*sourcedir), cfg_getstr(cfg, SOURCEDIR));
    }

    /* delete the conf file structure */
    cfg_free(cfg);
    return 0;
}


/*
 *  Initialises the pipeline (client-side method stub).
 *
 *  \param    *remoteobj  ID of the remote object
 *  \param     filter     Image filter which will be used
 *  \param    *sourcedir  Directory containing image sources.
 *  \param    *targetdir  Directory containing processed images
 *  \param    *successor  ID of the next plug-in in the pipeline
 *  \param    *ppmnode    Node with the running PPM
 *  \return               0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_initpipeline"
int rmiximgprocclient_initpipeline ( rmix_remoteref_t *remoteobj,
                                      unsigned int      filter,
                                      char             *sourcedir,
                                      char             *targetdir,
                                      char             *successor,
                                      char             *predecessor,
                                      char             *ppmnode)
{
    const void *inary[6];
    void       *outary[1];
    int         result;

    /* Prepare parameters. */
    inary[0] = &filter;
    inary[1] =  sourcedir;
    inary[2] =  targetdir;
    inary[3] =  successor;
    inary[4] =  predecessor;
    inary[5] =  ppmnode;

    outary[0] = &result;

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_INITPIPELINE_INDEX, outary, 1,
                         inary, 6))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return result;
```

```
}


/*
 *  Invokes the pipeline (client-side method stub).
 *
 *  \param     *remoteobj  ID of the remote object
 *  \return                0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_invokepipeline_oneway"
int rmiximgprocclient_invokepipeline_oneway (rmix_remoteref_t *remoteobj)
{
    /* Invoke remote object method. */
    if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_INVOKEPIPELINE_INDEX, NULL, 0))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}


/*
 *  Checks the availability of a plug-in (client-side method stub).
 *
 *  \param     *remoteobj  ID of the remote object
 *  \return                0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_availabilitycheck"
int rmiximgprocclient_availabilitycheck( rmix_remoteref_t *remoteobj)
{
    void        *outary[1];
    int          result;

    outary[0] = &result;

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_AVAILABILITYCHECK_INDEX, outary, 1,
                         NULL, 0))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}


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

```
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_updateimagecounter"
int rmiximgprocclient_updateimagecounter( rmix_remoteref_t *remoteobj,
                                          unsigned int      imagecounter)
{

    const void *inary[1];
    void       *outary[1];
    int         result;

    /* Prepare parameters. */
    inary[0] = &imagecounter;

    outary[0] = &result;

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_UPDATEIMAGECOUNTER_INDEX, outary,
                         1, inary, 1))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}


/*
 * Update the successor entry of the predecessor plug-in and
 * returns the image counter (client-side method stub).
 *
 * \param   *remoteobj     ID of the remote object
 * \param   *successor     New successor entry
 * \param   *imagecounter  The returned image counter from the predecessor
 * \return                 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_updatepredecessor"
int rmiximgprocclient_updatepredecessor( rmix_remoteref_t *remoteobj,
                                         char             *successor,
                                         unsigned int     *imagecounter)
{

    const void *inary[1];

    void       *outary[2];
    int         result;

    /* Prepare parameters. */
    inary[0] = successor;

    outary[0] = &result;
    outary[1] = &(*imagecounter);

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_UPDATEPREDECESSOR_INDEX, outary,
                         2, inary, 1))
    {
```

```
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}


/*
 *  Update the predecessor entry of the successor plug-in (client-side method
 *  stub).
 *
 *  \param    *remoteobj     ID of the remote object
 *  \param    *predecessor   New predecessor entry
 *  \return                  0 on success or -1 on any error
 */
#undef   __FUNC__
#define __FUNC__ "rmiximgprocclient_updatesuccessor"
int rmiximgprocclient_updatesuccessor( rmix_remoteref_t *remoteobj,
                                       char             *predecessor)
{

    const void *inary[1];

    void        *outary[1];
    int          result;

    /* Prepare parameters. */
    inary[0] = predecessor;

    outary[0] = &result;

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_UPDATESUCCESSOR_INDEX, outary,
                         1, inary, 1))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}


/*
 *  Accepts calls for repairing the pipeline in case of an error. (server-side
 *  stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
```

```
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmixppm_repairpipe_call"
RMIX_METHOD_CALL(rmixppm_repairpipe_call)
{
    int   result = 0;

    char *node;          /* predecessor plug-in */

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare input. */
    node = (char*)inary[0];
```

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

    /* Prepare output. */
    *(int*)(outary[0]) = result;
    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Tries to repair the pipeline in case of an error.
 *
 *  \param   *node  Node which cannot be accessed
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmixppm_repairpipe"
int rmixppm_repairpipe (char *node)
{
    int error;
    int ret;
    int i;
    int position;
    unsigned int      imagecounter;
    configlist_t     *tmpptrnodes;
    configlist_t     *plugin;
    rmix_remoteref_t *remoteobj;
    rmix_remoteref_t *remoteobj_pre;  /* reference to the predecessor plug-in */
    rmix_remoteref_t *remoteobj_suc;  /* reference to the successor plug-in   */


    /* Lock ppm pipe mutex. */
    if (0 != (error = pthread_mutex_lock(&ppm_data.pipemutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to lock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }

    PPM_PRINT((
        PPM_INFO(*****************************************)))
    PPM_PRINT((
        PPM_INFO(repair pipeline function called missing node = %s),node))

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

        /* check for available nodes */
```

```
        if (configlist_listsize( ppm_data.freenodes) != 0)
        {
            PPM_PRINT((
                PPM_INFO(%d free nodes avalaible),
                                configlist_listsize( ppm_data.freenodes)))

            tmpptrnodes = ppm_data.freenodes;

            /* get the name of the failed plug-in unit */
            ret = configlist_return_element( &plugin, position,
                                              ppm_data.dist_plugins);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(plug-in could not find plug-in to reload)))

                /* Unlock ppm pipe mutex. */
                if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                return -1;
            }

            PPM_PRINT((
                PPM_INFO(name of the plug-in to reload is %s),
                                                plugin->name))

            /* try to reload the failed plug-in component on one of the free
               nodes */
            ret = ppm_load_harnesskernel_distributed( &ppm_data.freenodes,
                                                       plugin, NULL);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(plug-in could not be reloaded)))

                /* Unlock ppm pipe mutex. */
                if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                return -1;
            }

            /* free memory */
            configlist_delete_list(&plugin);

            /* update the node with the reloaded plug-in in the list of used
            nodes and delete from the list of free nodes */
            if ( (configlist_reset_entry( ppm_data.freenodes->name, position,
                                          ppm_data.usednodes) != 0)
               || (configlist_delete_entry( 0, &ppm_data.freenodes) != 0) )
            {
                PPM_PRINT((
```

```
                PPM_WARN(plug−in could not add the new node to the list
                        of used nodes)))

            /* Unlock ppm pipe mutex. */
            if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
            {
                errno = error;
                PPM_PRINT((
                    PPM_WARN(unable to unlock ppm plug−in mutex)))
                harness_syserr();
                return −1;
            }
            return −1;
        }

        /* reinitialise the new loaded plug−in and its neighbours */

        /* move to the position of the reloaded node */
        tmpptrnodes = ppm_data.usednodes;
        for ( i=0; i<position; i++)
            tmpptrnodes = tmpptrnodes−>next;

        /* prepare the remote reference */
        ppm_createremoteref( &remoteobj, "1002", tmpptrnodes−>name);
        /* check if the reloaded is the first unit of the pipeline
           the first plug−in unit has to open the images */
        if(position==0)
        {
            ret = rmiximgprocclient_initpipeline( remoteobj, position,
                                                  ppm_data.sourcedir,
                                                  "nada",
                                                  tmpptrnodes−>next−>name,
                                                  "nada", ppm_data.ppmnode);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))
                rmix_remoteref_destroy( &remoteobj);
                /* Unlock ppm pipe mutex. */
                if (0 != (error = pthread_mutex_unlock(
                                      &ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug−in mutex)))
                    harness_syserr();
                    return −1;
                }
                return −1;
            }

            /* prepare the remote reference of the successor */
            ppm_createremoteref( &remoteobj_suc, "1002",
                                 tmpptrnodes−>next−>name);
            /* update the predecessor entry of the successor plug−in */
            ret = rmiximgprocclient_updatesuccessor( remoteobj_suc,
                                                     tmpptrnodes−>name);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))
                rmix_remoteref_destroy( &remoteobj_suc);
                /* Unlock ppm pipe mutex. */
```

```
                if (0 != (error = pthread_mutex_unlock(
                                        &ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                return -1;
            }
            rmix_remoteref_destroy( &remoteobj_suc);

            /* to prevent a possible image loss resend all images by
               reinvoking the pipeline */
            ret = rmiximgprocclient_invokepipeline_oneway(remoteobj);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not reinvoke pipeline)))
                rmix_remoteref_destroy( &remoteobj);
                /* Unlock ppm pipe mutex. */
                if (0 != (error = pthread_mutex_unlock(
                                        &ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                return -1;
            }
        }
        /* the last plug-in unit has to store the images */
        else if(position == (configlist_listsize(ppm_data.dist_plugins)-1))
        {
            ret = rmiximgprocclient_initpipeline( remoteobj, position,
                                        "nada",ppm_data.targetdir,
                                        "nada",
                                        tmpptrnodes->prev->name,
                                        ppm_data.ppmnode);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))
                rmix_remoteref_destroy( &remoteobj);
                /* Unlock ppm pipe mutex. */
                if (0 != (error = pthread_mutex_unlock(
                                        &ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                return -1;
            }

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

```

        /* update the successor entry of the predecessor plug-in and
           gets its image counter */
        ret = rmiximgprocclient_updatepredecessor( remoteobj_pre,
                                                   tmpptrnodes->name,
                                                   &imagecounter);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not initialize pipeline)))
            rmix_remoteref_destroy( &remoteobj_pre);
            /* Unlock ppm pipe mutex. */
            if (0 != (error = pthread_mutex_unlock(
                                    &ppm_data.pipemutex)))
            {
                errno = error;
                PPM_PRINT((
                    PPM_WARN(unable to unlock ppm plug-in mutex)))
                harness_syserr();
                return -1;
            }
            return -1;
        }

        /* update the image counter of the new loaded pipeline unit */
        ret = rmiximgprocclient_updateimagecounter( remoteobj,
                                                    imagecounter);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not initialize pipeline)))
            rmix_remoteref_destroy( &remoteobj);
            /* Unlock ppm pipe mutex. */
            if (0 != (error = pthread_mutex_unlock(
                                    &ppm_data.pipemutex)))
            {
                errno = error;
                PPM_PRINT((
                    PPM_WARN(unable to unlock ppm plug-in mutex)))
                harness_syserr();
                return -1;
            }
            return -1;
        }

        /* trigger the successor plug-in to resend its internal backup
           list to prevent an image loss */
        ret = rmiximgprocclient_sendworklist_oneway (remoteobj_pre);
        if (ret != 0)
        {
            PPM_PRINT((
                PPM_WARN(could not initialize pipeline)))
            rmix_remoteref_destroy( &remoteobj_pre);
            /* Unlock ppm pipe mutex. */
            if (0 != (error = pthread_mutex_unlock(
                                    &ppm_data.pipemutex)))
            {
                errno = error;
                PPM_PRINT((
                    PPM_WARN(unable to unlock ppm plug-in mutex)))
                harness_syserr();
                return -1;
            }
```

```
                return -1;
            }
            rmix_remoteref_destroy( &remoteobj_pre );

        }
        /* every plug-in unit in between has to forward the images and
           acknowledgments */
        else
        {
            ret = rmiximgprocclient_initpipeline( remoteobj, position,
                                                  "nada", "nada",
                                                  tmpptrnodes->next->name,
                                                  tmpptrnodes->prev->name,
                                                  ppm_data.ppmnode);
            if (ret != 0)
            {
                PPM_PRINT((
                    PPM_WARN(could not initialize pipeline)))
                rmix_remoteref_destroy( &remoteobj);
                /* Unlock ppm pipe mutex. */
                if (0 != (error = pthread_mutex_unlock(
                                    &ppm_data.pipemutex)))
                {
                    errno = error;
                    PPM_PRINT((
                        PPM_WARN(unable to unlock ppm plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                return -1;
            }

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

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

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

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

```
   gets its image counter */
ret = rmiximgprocclient_updatepredecessor( remoteobj_pre,
                                           tmpptrnodes->name,
                                           &imagecounter);
if (ret != 0)
{
    PPM_PRINT((
        PPM_WARN(could not initialize pipeline)))
    rmix_remoteref_destroy( &remoteobj_pre);
    /* Unlock ppm pipe mutex. */
    if (0 != (error = pthread_mutex_unlock(
                            &ppm_data.pipemutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to unlock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }
    return -1;
}

/* set the image counter of the new loaded pipeline unit */
ret = rmiximgprocclient_updateimagecounter( remoteobj,
                                            imagecounter);
if (ret != 0)
{
    PPM_PRINT((
        PPM_WARN(could not initialize pipeline)))
    rmix_remoteref_destroy( &remoteobj);
    /* Unlock ppm pipe mutex. */
    if (0 != (error = pthread_mutex_unlock(
                            &ppm_data.pipemutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to unlock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }
    return -1;
}

/* trigger the predecessor to resend its internal backup list
   to prevent a possible image loss */
ret = rmiximgprocclient_sendworklist_oneway (remoteobj_pre);
if (ret != 0)
{
    PPM_PRINT((
        PPM_WARN(could not initialize pipeline)))
    rmix_remoteref_destroy( &remoteobj_pre);
    /* Unlock ppm pipe mutex. */
    if (0 != (error = pthread_mutex_unlock(
                            &ppm_data.pipemutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to unlock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }
    return -1;
}
```

```
                rmix_remoteref_destroy( &remoteobj_pre);

            }
            rmix_remoteref_destroy( &remoteobj);

        }
        else
        {
            PPM_PRINT((
                PPM_INFO(no free node(s) avalaible)))
        }
    }
    else
    {
        PPM_PRINT((
            PPM_INFO(missing node %s not found in list), node))
    }

    PPM_PRINT((
        PPM_INFO(****************************************)))

    /* Unlock ppm pipe mutex. */
    if (0 != (error = pthread_mutex_unlock(&ppm_data.pipemutex)))
    {
        errno = error;
        PPM_PRINT((
            PPM_WARN(unable to unlock ppm plug-in mutex)))
        harness_syserr();
        return -1;
    }

    return 0;
}


/*
 *  Sends again the worklist to the reloaded plug-in (client-side method stub).
 *
 *  \param    *remoteobj  ID of the remote object
 *  \return               0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_sendworklist_oneway"
int rmiximgprocclient_sendworklist_oneway (rmix_remoteref_t *remoteobj)
{
    /* Invoke remote object method. */
    if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_SENDWORKLIST_INDEX, NULL, 0))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}

/***************************************************************************
 *
 * END OF FILE
 *
 ***************************************************************************/
```

A.3.2.3. Header File for the PPM Utility Library

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

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


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


/*****************************************************************************
 *
 * Macros
 *
 *****************************************************************************/

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


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


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


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

```

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


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


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


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


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


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


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


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


/** \def    INPUT
```

```
 * \brief Input sectionin a configuration file is converted into a linked
 *        list.
 */
#define INPUT 3

/** \def   NODESECTION
 * \brief Name of the node section in the configuration file.
 */
#define NODESECTION  "nodelist"

/** \def   NODEENTRY
 * \brief Name of the entry in the configuration file node section.
 */
#define NODEENTRY    "adresses"

/** \def   RSECTION
 * \brief Name of the replicated plug-in section in the configuration file.
 */
#define RSECTION     "replicated"

/** \def   RENTRY
 * \brief Name of the entry in the configuration file replicated section.
 */
#define RENTRY       "plugins"

/** \def   DSECTION
 * \brief Name of the distributed plug-in section in the configuration file.
 */
#define DSECTION     "distributed"

/** \def   DENTRY
 * \brief Name of the entry in the configuration file distributed section.
 */
#define DENTRY       "plugins"

/** \def   INPUTSECTION
 * \brief Name of the input section in the configuration file.
 */
#define INPUTSECTION "input"

/** \def   INPUTENTRY
 * \brief Name of the entry in the configuration file input section.
 */
#define INPUTENTRY   "files"

/** \def   PPMMODE
 * \brief Selection parallel plug-in mode (1) or distributed parallel plug-in
 *        mode (0).
 */
#define PPMMODE      "mode"

/** \def   PPMNODE
 * \brief Node of the Parallel Plug-in Manager.
```

```
 */
#define PPMNODE        "ppmnode"


/***************************************************************************
 *
 * Includes
 *
 ***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <confuse.h>


/***************************************************************************
 *
 * Data Types
 *
 ***************************************************************************/

/** \struct configlist_t
 *  \brief  Element of the linked list for storing configuration information.
 *
 *  Each element contains a name for storing ip-addresses of hosts or the name
 *  of a plug-in and a pointer to the next and previous element in the list.
 */
typedef struct configlist
{
    char    *name;                 /* ip-address or name of plug-in    */
    struct configlist *next;       /* pointer to the next list element */
    struct configlist *prev;       /* pointer to the prev list element */
}configlist_t;


/***************************************************************************
 *
 * Function Prototypes
 *
 ***************************************************************************/

/** \fn     int configuration_read_file( configlist_t **nodelist, \
                                         configlist_t **replicatedlist, \
                                         configlist_t **distributedlist, \
                                         configlist_t **inputlist \
                                         int           *mode, \
                                         char         **ppmnode);
 *  \brief  Reads the config file, information is stored in linked lists.
 *
 *  \param  **nodelist         List pointer for the node name(s)
 *  \param  **replicatedlist   List pointer for replicated plug-in name(s)
 *  \param  **distributedlist  List pointer for distributed plug-in name(s)
 *  \param  **inputlist        List pointer for input file(s)
 *  \param   *mode             Replicated or distributed parallel plug-in mode
 *  \param  **ppmnode          Node of the PPM
 *  \return                    0 on success or -1 on any error
 */
int configuration_read_file( configlist_t **nodelist,
                             configlist_t **replicatedlist,
                             configlist_t **distributedlist,
                             configlist_t **inputlist,
```

```
                                int            *mode,
                                char          **ppmnode);


/** \fn      int configuration_validate_nodelist( cfg_t      *cfg, \
                                                 cfg_opt_t *opt);
 *  \brief   Validates the node section in the configuration file.
 *
 *  \param   *cfg Pointer to the configuration file structure
 *  \param   *opt Name of the section to validate
 *  \return       0 on success or -1 on any error
 */
int configuration_validate_nodelist( cfg_t      *cfg,
                                     cfg_opt_t *opt);


/** \fn      int configlist_insert_element( char           *element, \
                                           configlist_t **list_pointer);
 *  \brief   Insert an element in a list.
 *
 *  \param    *element      New list element name
 *  \param   **list_pointer Pointer to the linked list
 *  \return                 0 on success or -1 on any error
 */
int configlist_insert_element( char           *element,
                               configlist_t **list_pointer);


/** \fn      void configuration_conver2list( cfg_t          *cfg, \
                                            configlist_t **list_pointer, \
                                            int             type);
 *  \brief   Converts a section of the configuration file structure into a list.
 *
 *  \param    *cfg          Pointer to the structure of the configuration file
 *  \param   **list_pointer Pointer to the new linked list
 *  \param    type          Defines the section of the configuration file
 *  \return                 0 on success or -1 on any error
 */
int configuration_conver2list( cfg_t          *cfg,
                               configlist_t **list_pointer,
                               int             type);


/** \fn      void configlist_delete_list( configlist_t **list_pointer);
 *  \brief   Deletes the linked list defined by the pointer.
 *
 *  \param   **list_pointer Pointer to the linked list
 */
void configlist_delete_list( configlist_t **list_pointer);


/** \fn      void configlist_print_list( configlist_t *list_pointer);
 *  \brief   Prints the list elements.
 *
 *  \param   *list_pointer Pointer to the linked list
 */
void configlist_print_list( configlist_t *list_pointer);


/** \fn      int configlist_listsize( configlist_t *list_pointer);
 *  \brief   Return the number of list elements.
 *
 *  \param   *list_pointer Pointer to the linked list
```

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


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


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


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


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


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

```
*/
int configlist_delete_entry( int              position,
                             configlist_t **list_pointer);


#endif
/***************************************************************************
 *
 * END OF FILE
 *
 ***************************************************************************/
```

A.3.2.4. Source File for the PPM Utility Library

```
/***************************************************************************
 *
 * Source file for the Integral-Loader module.
 * Copyright (c) Ronald Baumann.
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 ***************************************************************************/

/** \file  readconf.c
 *  \brief Sorce file for plug-in loader utils library.
 *
 *  The plug-in loader utils library contains functions which are used by
 *  parallel plug-in manager.
 *  It supports the reading of a configuration file and the storing of the read
 *  data in lists.
 */


/***************************************************************************
 *
 * Includes
 *
 ***************************************************************************/
#include "readconf.h"


/***************************************************************************
 *
 * Functions
 *
 ***************************************************************************/

/*
 *  Reads the config file, information is stored in linked lists.
 *
 *  \param  **nodelist         List pointer for the node name(s)
 *  \param  **replicatedlist   List pointer for replicated plug-in name(s)
 *  \param  **distributedlist  List pointer for distributed plug-in name(s)
 *  \param  **inputlist        List pointer for input file(s)
 *  \param   *mode             Replicated or distributed parallel plug-in mode
```

```
 *  \param   **ppmnode          Node of the PPM
 *  \return                     0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "configuration_read_file"
int configuration_read_file( configlist_t **nodelist,
                             configlist_t **replicatedlist,
                             configlist_t **distributedlist,
                             configlist_t **inputlist,
                             int           *mode,
                             char         **ppmnode)
{
    /* defines the node section in the conf file */
    cfg_opt_t nodelist_opts[] =
    {
        CFG_STR_LIST( NODEENTRY, , CFGF_NONE),
        CFG_END()
    };

    /* defines the replicated plug-in section in the conf file */
    cfg_opt_t replicated_opts[] =
    {
        CFG_STR_LIST( RENTRY, , CFGF_NONE),
        CFG_END()
    };

    /* defines the distributed plug-in section in the conf file */
    cfg_opt_t distributed_opts[] =
    {
        CFG_STR_LIST( DENTRY, , CFGF_NONE),
        CFG_END()
    };

    /* defines the input section in the conf file */
    cfg_opt_t input_opts[] =
    {
        CFG_STR_LIST( INPUTENTRY, , CFGF_NONE),
        CFG_END()
    };

    /* build the conf file structure */
    cfg_opt_t opts[] =
    {
        CFG_SEC( NODESECTION, nodelist_opts, CFGF_NONE),
        CFG_SEC( RSECTION, replicated_opts, CFGF_NONE),
        CFG_SEC( DSECTION, distributed_opts, CFGF_NONE),
        CFG_SEC( INPUTSECTION, input_opts, CFGF_NONE),
        CFG_INT( PPMMODE, 0, CFGF_NONE),
        CFG_STR( PPMNODE, NULL, CFGF_NONE),
        CFG_END()
    };

    cfg_t *cfg;

    READCONF_PRINT((
        READCONF_INFO(start reading configuration file)))

    /* initialize the conf file structure */
    cfg = cfg_init(opts, CFGF_NONE);

    /* initialize the validation function */
    cfg_set_validate_func(cfg, NODESECTION, configuration_validate_nodelist);

```

```
    /* read the conf file */
    if(cfg_parse(cfg, CONFFILE) == CFG_PARSE_ERROR)
        return -1;

    /* convert the conf file section to lists */
    if ( configuration_conver2list( cfg, nodelist, NODES) == -1)
        return -1;
    if ( configuration_conver2list( cfg, replicatedlist, REPLICATED) == -1)
        return -1;
    if ( configuration_conver2list( cfg, distributedlist, DISTRIBUTED) == -1)
        return -1;
    if ( configuration_conver2list( cfg, inputlist, INPUT) == -1)
        return -1;

    /* read the node, on which the PPM runs */
    *mode = cfg_getint(cfg, PPMMODE);

    *ppmnode = (char*)malloc( sizeof(char) * (strlen(cfg_getstr(cfg, PPMNODE))
                                                  + 1));
    if ((*ppmnode) == NULL)
    {
        READCONF_PRINT((
            READCONF_WARN(could not allocate memory)))
        return -1;
    }
    strcpy( (*ppmnode), cfg_getstr(cfg, PPMNODE));

    /* delete the conf file structure */
    cfg_free(cfg);
    return 0;
}


/*
 *  Validates the node section in the configuration file.
 *
 *  \param  *cfg Pointer to the configuration file structure
 *  \param  *opt Name of the section to validate
 *  \return      0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "configuration_validate_nodelist"
int configuration_validate_nodelist( cfg_t     *cfg,
                                     cfg_opt_t *opt)
{
    /* get the last parsed section */
    cfg_t *sec = cfg_opt_getnsec(opt, cfg_opt_size(opt) - 1);

    /* validate that a "string" option is set in the section */
    if(cfg_size(sec, NODEENTRY) > 0 && cfg_getstr(sec, NODEENTRY) != NULL)
        return CFG_SUCCESS;

    /* otherwise the option is not set at all, log a message and return error */
    cfg_error(cfg, "missing adresses option in nodelist section");
    READCONF_PRINT((
        READCONF_WARN(missing adresses option in nodelist section of
                      configuration file)))
    return CFG_PARSE_ERROR;
}


/*
 *  Converts a section of the configuration file structure into a list.
```

```
 *
 * \param    *cfg           Pointer to the structure of the configuration file
 * \param  **list_pointer Pointer to the new linked list
 * \param    type           Defines the section of the configuration file
 * \return                  0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "configuration_conver2list"
int configuration_conver2list( cfg_t          *cfg,
                               configlist_t **list_pointer,
                               int             type)
{
    cfg_t *section;
    int i;
    int ret;

    /* which section is converted? */
    switch(type)
    {
        case(NODES):
        {
            section = cfg_getsec(cfg, NODESECTION);
            /* control if the section contains information */
            if (cfg_getstr(section, NODEENTRY) == NULL)
            {
                (*list_pointer) = NULL;
                return -1;
            }
            /* create list */
            for( i = 0; i < cfg_size(section, NODEENTRY); i++)
            {
                ret = configlist_insert_element(
                           cfg_getnstr(section, NODEENTRY, i), list_pointer);
                if (ret == -1)
                    return -1;
            }
            break;
        }
        case(REPLICATED):
        {
            section = cfg_getsec(cfg, RSECTION);
            /* control if the section contains information */
            if (cfg_getstr(section, RENTRY) == NULL)
            {
                (*list_pointer) = NULL;
                return -1;
            }
            /* create list */
            for( i = 0; i < cfg_size(section, RENTRY); i++)
            {
                ret = configlist_insert_element(
                           cfg_getnstr(section, RENTRY, i), list_pointer);
                if (ret == -1)
                    return -1;
            }
            break;
        }
        case(DISTRIBUTED):
        {
            section = cfg_getsec(cfg, DSECTION);
            /* control if the section contains information */
            if (cfg_getstr(section, DENTRY) == NULL)
            {
```

```
                (*list_pointer) = NULL;
                return -1;
            }
            /* create list */
            for( i = 0; i < cfg_size(section, DENTRY); i++)
            {
                ret = configlist_insert_element(
                           cfg_getnstr(section, DENTRY, i), list_pointer);
                if (ret == -1)
                    return -1;
            }
            break;
        }
        case(INPUT):
        {
            section = cfg_getsec(cfg, INPUTSECTION);
            /* control if the section contains information */
            if (cfg_getstr(section, INPUTENTRY) == NULL)
            {
                (*list_pointer) = NULL;
                return -1;
            }
            /* create list */
            for( i = 0; i < cfg_size(section, INPUTENTRY); i++)
            {
                ret = configlist_insert_element(
                           cfg_getnstr(section, INPUTENTRY, i), list_pointer);

                if (ret == -1)
                    return -1;
            }
            break;
        }
    }
    return 0;
}


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

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

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

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


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

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

        tmp_element->next = new_element;
        new_element->prev = tmp_element;
        new_element->next = NULL;
    }

    return 0;
}


/*
 *  Prints the list elements.
 *
 *  \param  *list_pointer Pointer to the linked list
 */
#undef  __FUNC__
#define __FUNC__ "configlist_print_list"
void configlist_print_list( configlist_t *list_pointer)
{
    configlist_t *current_element;

    current_element = list_pointer;
    while ( current_element != NULL )
    {
        READCONF_PRINT((
            READCONF_INFO(name = %s\n), current_element->name))

        current_element = current_element->next;
    }
}


/*
```

```
 *  Deletes the linked list defined by the pointer.
 *
 *  \param  **list_pointer Pointer to the linked list
 */
#undef  __FUNC__
#define __FUNC__ "configlist_delete_list"
void configlist_delete_list( configlist_t **list_pointer)
{
    configlist_t *tmp_pointer;

    while( *list_pointer != NULL )
    {
        tmp_pointer = *list_pointer;
        *list_pointer = (*list_pointer)->next;

        /* delete name of the element */
        FREE(tmp_pointer->name);

        /* delete element */
        FREE(tmp_pointer);
    }
    *list_pointer = NULL;
}


/*
 *  Return the number of list elements.
 *
 *  \param  *list_pointer Pointer to the linked list
 *  \return               Number of list elements
 */
#undef  __FUNC__
#define __FUNC__ "configlist_listsize"
int configlist_listsize( configlist_t *list_pointer)
{
    configlist_t *current_element;
    int counter = 0;

    current_element = list_pointer;
    while ( current_element != NULL )
    {
        current_element = current_element->next;
        counter ++;
    }
    return counter;
}


/*
 *  Checks if all names entries != NULL.
 *
 *  \param  *list_pointer Pointer to the linked list
 *  \return               0 if all name entries != NULL otherwise -1
 */
#undef  __FUNC__
#define __FUNC__ "configlist_check_entries"
int configlist_check_entries( configlist_t *list_pointer)
{
    configlist_t *current_element;

    current_element = list_pointer;
    while ( current_element != NULL )
    {
```

```
        if ( current_element->name == NULL )
            return -1;
        current_element = current_element->next;
    }
    return 0;
}


/*
 *  Finds the position of an entry.
 *
 *  \param  *char          Entry to find
 *  \param  *list_pointer  Pointer to the linked list
 *  \return                The position or otherwise -1
 */
#undef  __FUNC__
#define __FUNC__ "configlist_find_elementposition"
int configlist_find_elementposition( char          *entry,
                                     configlist_t *list_pointer)
{
    configlist_t *current_element;
    int position = 0;

    current_element = list_pointer;
    while ( current_element != NULL )
    {
        if ( strcmp( current_element->name, entry) == 0 )
            return position;
        current_element = current_element->next;
        position++;
    }

    return -1;
}


/*
 *  Returns an element found at the specific position
 *
 *  \param  **element      Found list element
 *  \param    position     Position of the element in the list
 *  \param   *list_pointer Pointer to the linked list
 *  \return                0 on success otherwise -1
 */
#undef  __FUNC__
#define __FUNC__ "configlist_return_element"
int configlist_return_element( configlist_t **element,
                               int             position,
                               configlist_t  *list_pointer)
{
    configlist_t *current_element;
    int i;

    current_element = list_pointer;

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

```

```
    /* move forward to position */
    for ( i=0; i<position; i++)
    {
        current_element = current_element->next;
    }

    (*element) = (configlist_t*)malloc( sizeof(configlist_t));
    if ( (*element) == NULL )
    {
        READCONF_PRINT((
            READCONF_WARN(allocating of memory for found list element not
                          possible)))
        return -1;
    }

    /* allocate memory for the name string of the element */
    (*element)->name =(char*)malloc( sizeof(char) *
                                     (strlen(current_element->name) + 1));
    if ( (*element)->name == NULL )
    {
        READCONF_PRINT((
            READCONF_WARN(allocating of memory for the name string of the
                          list element not possible)))
        return -1;
    }

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

    return 0;
}


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

    tmp_pointer = list_pointer;

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

    tmp_pointer = list_pointer;
```

```
    for (i=0; i<position; i++)
        tmp_pointer = tmp_pointer->next;

    FREE(tmp_pointer->name);

    /* allocate memory for the name string of the new element */
    tmp_pointer->name = (char*)malloc(sizeof(char)*(strlen(entry)+1));
    if ( tmp_pointer->name == NULL )
    {
        READCONF_PRINT((
            READCONF_WARN(allocating of memory for the name string not
                possible)))
        return -1;
    }

    /* store the information */
    strcpy( tmp_pointer->name, entry);

    return 0;
}


/*
 *  Deletes the element at the position.
 *
 *  \param   *int           Position of the element.
 *  \param   **list_pointer Pointer to the linked list
 *  \return                 0 on success otherwise -1
 */
#undef  __FUNC__
#define __FUNC__ "configlist_delete_entry"
int configlist_delete_entry( int            position,
                             configlist_t **list_pointer)
{
    configlist_t *tmp_pointer;
    configlist_t *del_element;
    int i=0;

    tmp_pointer = *list_pointer;

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

    /* if it is the first element in the list change the pointer to the list*/
    if ( position == 0 )
    {
        tmp_pointer = *list_pointer;
        *list_pointer = (*list_pointer)->next;
        if ((*list_pointer) != NULL)
            (*list_pointer)->prev = NULL;

        /* delete name of the element */
        FREE(tmp_pointer->name);

        /* delete element */
        FREE(tmp_pointer);
    }
    else
```

```
    {
        tmp_pointer = *list_pointer;

        for (i=0; i<position-1; i++)
            tmp_pointer = tmp_pointer->next;

        del_element = tmp_pointer->next;

        /* if it was the last element in the list */
        if (del_element->next == NULL)
        {
            tmp_pointer->next = NULL;
        }
        else
        {
            tmp_pointer->next = del_element->next;
            del_element->next->prev = tmp_pointer;
        }

        FREE(del_element->name);
        FREE(del_element);
    }

    return 0;
}

/*****************************************************************************
 *
 * END OF FILE
 *
 *****************************************************************************/
```

A.3.3. Parallel Plug-in for Integral Computation

A.3.3.1. Header File

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

/** \file integral.h
 *  \brief Header file for integral module.
 *
```

```
 * 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.
 */

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


/****************************************************************************
 *
 * Macros
 *
 ****************************************************************************/

/** \def    LIBINTEGRAL_PACKAGE_NAME
 *  \brief Package name (unquoted).
 */
#define LIBINTEGRAL_PACKAGE_NAME montecarlo


/** \def    LIBINTEGRAL_PACKAGE_VERSION
 *  \brief Package version (unquoted).
 */
#define LIBINTEGRAL_PACKAGE_VERSION 1.1


/** \def    LIBINTEGRAL_PACKAGE_RELEASE
 *  \brief Package release (unquoted).
 */
#define LIBINTEGRAL_PACKAGE_RELEASE 0


/** \def    LIBINTEGRAL_PACKAGE_BUGREPORT
 *  \brief Package bug report e-mail (unquoted).
 */
#define LIBINTEGRAL_PACKAGE_BUGREPORT baumannr@ornl.gov


/** \def    LIBINTEGRAL_VERSION_CURRENT
 *  \brief Version current (unquoted).
 */
#define LIBINTEGRAL_VERSION_CURRENT 0


/** \def    LIBINTEGRAL_VERSION_REVISION
 *  \brief Version revision (unquoted).
 */
#define LIBINTEGRAL_VERSION_REVISION 0


/** \def    LIBINTEGRAL_VERSION_AGE
 *  \brief Version age (unquoted).
 */
#define LIBINTEGRAL_VERSION_AGE 0


/** \def    LIBINTEGRAL_VERSION_FIRST
 *  \brief Version first (unquoted).
 */
#define LIBINTEGRAL_VERSION_FIRST 0

```

```c
/** \def    LIBINTEGRAL_VERSION
 *  \brief Version (unquoted).
 */
#define LIBINTEGRAL_VERSION 0.0.0

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

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

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

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

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

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

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

/* Flag for <harness.0/harness.h> header. */
#ifndef     HARNESS_HARNESS_H
#ifndef     HAVE_HARNESS_0_HARNESS_H
#define      HAVE_HARNESS_0_HARNESS_H 1
#endif   /* HAVE_HARNESS_0_HARNESS_H */
```

```
#endif  /* HARNESS_HARNESS_H */


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


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


/** \def   TESTMODE
 *  \brief 0 = normal parallel plug-in mode, 1 = testmode without communication
 *         by using the testfile.
 */
#define TESTMODE 0


/*****************************************************************************
 *
 * Includes
 *
 *****************************************************************************/

/* Include <harness.0/harness.h> header. */
#if       HAVE_HARNESS_0_HARNESS_H
#include      <harness.0/harness.h>
#endif /* HAVE_HARNESS_0_HARNESS_H */

#include <math.h>
#include "rmixintegral.h"


/*****************************************************************************
 *
 * Data Types
 *
 *****************************************************************************/

/** \struct integral_t
 *  \brief Integral plug-in "class".
 *
 *  This structure contains the public plug-in data and function pointers.
 *  It also provides information about the version of the plug-in.
 */
typedef struct
{
    struct
    {
        const unsigned int current;     /* current version        */
        const unsigned int revision;    /* current revision       */
        const unsigned int age;         /* version age            */
        const unsigned int first;       /* first supported version */
    }version;                           /* plug-in version        */
}integral_t;

```

```
/****************************************************************************
 *
 * Function Prototypes
 *
 ****************************************************************************/

/** \fn      int integral_init_rmix();
 *  \brief   Initializes the necessary rmix parameter and exports the plug-in
 *           communication interface.
 *
 *  \return 0 on success or -1 on any error
 */
int integral_init_rmix();


/*  \fn      int integral_fini_rmix();
 *  \brief   Finalizes rmix and unexports the plug-in.
 *
 *  \return 0 on success or -1 on any error
 */
int integral_fini_rmix();


/** \fn      RMIX_METHOD_CALL(rmixintegral_integration_call);
 *  \brief   Invokes the integration (server-side method stub).
 *
 *  \param   object   The local object.
 *  \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
 *                    dynamically or explicitly set to NULL by this function. In
 *                    the case of variable arrays, the length value is allocated
 *                    before the call and is set to 0 if its variable array is
 *                    NULL.
 *  \param   outcnt   The output values count.
 *  \param   inary    The input values array. The values array and its containing
 *                    values are not modified or deallocated by this call.
 *  \param   incnt    The input values count.
 *  \return           0 on success or -1 on any error with errno set
 *                    appropriately.
 */
RMIX_METHOD_CALL(rmixintegral_integration_call);


/** \fn      int rmixintegral_integration ( double        *result, \
                                           unsigned int  iterations, \
                                           double        lowerbound, \
                                           double        upperbound, \
                                           unsigned int  numcoeffs, \
                                           double        *coeffs);
 *  \brief   Calculates an integral with the received data and the Monte Carlo
 *           approach.
 *
 *  \param    *result       Returns the calculated integral
 *  \param     iterations   Number of random Monte Carlo numbers
 *  \param     lowerbound   Lower boundary of the integration interval
 *  \param     upperbound   Upper boundary of the integration interval
 *  \param     numcoeffs    Number of coefficients
 *  \param    *coeffs       Array with coefficients
 *  \return                 0 on success or -1 on any error
 */
int rmixintegral_integration ( double        *result,
                               unsigned int  iterations,
```

```
                                double         lowerbound,
                                double         upperbound,
                                unsigned int   numcoeffs,
                                double        *coeffs);


/** \fn     double integral_func_value( unsigned int  num_coeffs, \
                                        double       *coeffs, \
                                        double        x_value);
 *  \brief  Calculates a function value of a specified function.
 *
 *  \param   num_coeffs  Number of coefficients of the functions.
 *  \param  *coeffs      Array with the coefficients
 *  \param   x_value     Value, which is inserted into the function
 *  \return              The function value.
 */
double integral_func_value( unsigned int  num_coeffs,
                            double       *coeffs,
                            double        x_value);


/** \fn     double integral_random_number( double bottom, \
                                           double top);
 *  \brief  Generates a random number in a certain interval.
 *
 *  \param  bottom  Lower boundary of the interval.
 *  \param  top     Upper boundary of the interval
 *  \return         The random number
 */
double integral_random_number( double bottom,
                               double top);


/*********************************************************************************
 *
 *  The following section contains functions for testing the plug-in without
 *  the use of communication functions. A local input file is read and a
 *  integral is calculated.
 *
 *********************************************************************************/

/** \fn     int integral_calc_integral();
 *  \brief  Calculates the integral of a given function in a certain interval.
 *
 *  \return 0 on success or -1 on any error
 */
int integral_calc_integral();


/** \fn     int integral_read_inputfile( char          *filename, \
                                        unsigned int  *number, \
                                        double       **coeffs, \
                                        int           *iterations, \
                                        double        *bottom, \
                                        double        *top);
 *  \brief  Reads the input file, which contains the needed information.
 *
 *  \param  *filename    Name of the input file
 *  \param  *number      Returns the number of coefficients
 *  \param  **coeffs     Returns the array of coefficients
 *  \param  *iterations  Returns the amount of random numbers, which are used
 *  \param  *bottom      Returns the lower boundary of the interval
 *  \param  *top         Returns the upper boundary of the interval
```

```
 *  \return             0 on success or -1 on any error
 */
int integral_read_inputfile( char          *filename,
                             unsigned int  *number,
                             double       **coeffs,
                             int           *iterations,
                             double        *bottom,
                             double        *top);


/***************************************************************************
 *
 * Data
 *
 ***************************************************************************/

/** \var   extern integral_t integral
 *  \brief Integral plug-in "object".
 *
 *  An external "object" of the Integral plug-in "class".
 */
extern integral_t integral;

#endif /* INTEGRAL_INTEGRAL_H */


/***************************************************************************
 *
 * END OF FILE
 *
 ***************************************************************************/
```

A.3.3.2. Source File

```
/***************************************************************************
 *
 * Source file for the Integral module.
 * Copyright (c) Ronald Baumann.
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 ***************************************************************************/

/** \file  integral.c
 *  \brief Sorce file for integral module.
 *
 *  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.
 */


/***************************************************************************
 *
 * Includes
```

```
 *
 ***************************************************************************/

/* Main module header file. */
#include "integral.h"


/***************************************************************************
 *
 * Data Types
 *
 ***************************************************************************/

/** \struct integral_data_t
 *  \brief  Integral plug-in module data type.
 *
 *  Contains the mutex and the instances handler.
 */
typedef struct
{
    pthread_mutex_t  mutex;      /* mutex                */
    unsigned int     count;      /* instances array count */
    struct
    {
        unsigned int handle;     /* instance handle      */
    }*instances;                 /* instances array      */
}integral_data_t;


/** \struct integral_rmix_data_t
 *  \brief  Data for rmix communication functions.
 */
typedef struct
{
    rmix_localref_t *localref_harness;     /* exported Harness kernel  */
    rmix_localref_t *localref_integral;    /* exported integral object */
    unsigned int     handle_rmix;          /* handle for rmix plug-in  */
}integral_rmix_data_t;


/***************************************************************************
 *
 * Function Prototypes
 *
 ***************************************************************************/

/** \fn      HARNESS_PLUGINS_INIT(integral_init);
 *  \brief  Initializes the integral plug-in.
 *
 *  \param  handle The plug-in instance handle.
 *  \return 0 on success or -1 on any error with errno set appropriately.
 */
HARNESS_PLUGINS_INIT(integral_init);


/** \fn      HARNESS_PLUGINS_FINI(integral_fini);
 *  \brief  Finalizes the integral plug-in.
 *
 *  \param  handle The plug-in instance handle.
 *  \return 0 on success or -1 on any error with errno set appropriately.
 */
HARNESS_PLUGINS_FINI(integral_fini);
```

```
/*****************************************************************************
 *
 * Data
 *
 *****************************************************************************/

/** \var    const rmix_method_t rmixintegral_methods[RMIXINTEGRAL_METHODS_COUNT]
 *  \brief Server-side method descriptors for rmix integral.
 */
const rmix_method_t rmixintegral_methods[RMIXINTEGRAL_METHODS_COUNT] =
    RMIXINTEGRAL_METHODS;


/** \var    const rmix_interface_t rmixintegral_interface
 *  \brief Server-side interface for rmix integral.
 */
const rmix_interface_t rmixintegral_interface =
{
    RMIXINTEGRAL_METHODS_COUNT,             /* method descriptor count */
    (rmix_method_t*)rmixintegral_methods   /* method descriptor array */
};


/** \var    integral_data_t integral_data
 *  \brief Integral plug-in module data.
 *
 *  Includes the mutex for instances handle and the instances array where the
 *  handles are stored.
 */
integral_data_t integral_data =
{
    PTHREAD_MUTEX_INITIALIZER, /* mutex                */
    0,                         /* instances array count */
    NULL                       /* instances array       */
};


/** \var    integral_rmix_data_t integral_rmix_data
 *  \brief Data for rmix communication functions.
 *
 *  Includes the local references to the exported objects.
 */
integral_rmix_data_t integral_rmix_data =
{
    NULL,       /* exported Harness kernel  */
    NULL,       /* exported integral object */
    0           /* handle for rmix plug-in  */
};


/** \var    integral_t integral
 *  \brief HIntegral plug-in "object".
 *
 *  Contains the version of the library and possible public data and function
 *  pointers.
 */
integral_t integral =
{
    {
        LIBINTEGRAL_VERSION_CURRENT,    /* current version         */
        LIBINTEGRAL_VERSION_REVISION,   /* current revision        */
        LIBINTEGRAL_VERSION_AGE,        /* version age             */
```

```
        LIBINTEGRAL_VERSION_FIRST        /* first supported version */
    }                                    /* plug-in version         */
};

/***************************************************************************
 *
 * Functions
 *
 ***************************************************************************/

/*
 * Initializes the Integral plug-in.
 *
 * handle = The plug-in instance handle.
 * return = 0 on success or -1 on any error with errno set appropriately.
 */
#undef  __FUNC__
#define __FUNC__ "integral_init"
HARNESS_PLUGINS_INIT(integral_init)
{
    int            error;
    void          *instances;
    unsigned int   index;

    INTEGRAL_PRINT((
        INTEGRAL_INFO(libintegral is starting)))

    /* Lock integral plug-in mutex. */
    if (0 != (error = pthread_mutex_lock(&integral_data.mutex)))
    {
        errno = error;
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to lock integral plug-in mutex)))
        harness_syserr();
        return -1;
    }

    /* Search for handle in instances array. */
    for (index = 0; index < integral_data.count; index++)
    {
        if (integral_data.instances[index].handle == handle)
        {
            INTEGRAL_PRINT((
                INTEGRAL_WARN(handle is already in instances array)))
            harness_syserr();

            /* Unlock integral plug-in mutex. */
            if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
            {
                errno = error;
                INTEGRAL_PRINT((
                    INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
                harness_syserr();
            }

            return -1;
        }
    }

    /* Increase instances array. */
    index = integral_data.count;
    integral_data.count++;
```

```
    /* Reallocate instances array. */
    if (NULL == (instances = realloc( integral_data.instances,
                                      integral_data.count *
                              sizeof( integral_data.instances[0]))))
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to reallocate instances array)))
        /* Unlock integral plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
        {
            errno = error;
            INTEGRAL_PRINT((
                INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
            harness_syserr();
        }
        return -1;
    }

    integral_data.instances = instances;
    /* Save instances array entry. */
    integral_data.instances[index].handle = handle;
    /* Check for first initialization. */
    if (1 == integral_data.count)
    {
        /**************************/
        /* PUT YOUR INIT CODE HERE */
        /**************************/

        /* test mode is selected, the plug-in will read in an input file
           and compute an integral, otherwise it is part of the parallel
           plug-in */
        if (!TESTMODE)
        {
            /* if the communication interface cannot be exported terminate
               the plug-in */
            if (integral_init_rmix() != 0)
            {
                if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
                {
                    errno = error;
                    INTEGRAL_PRINT((
                        INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
                    harness_syserr();
                    return -1;
                }
                harness_kernel_shutdown();
            }
        }
        /* For testing the plug-in functions without using rmix functions */
        else
        {
            INTEGRAL_PRINT((
                INTEGRAL_INFO(running in test mode)))
            integral_calc_integral();
            harness_kernel_shutdown();
        }
    }

    /* Unlock integral plug-in mutex. */
    if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
    {
        errno = error;
        INTEGRAL_PRINT((
```

```
            INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
        harness_syserr();
        return -1;
    }
    return 0;
}


/*
 * Finalizes the integral plug-in.
 *
 * handle = The plug-in instance handle.
 * return = 0 on success or -1 on any error with errno set appropriately.
 */
#undef  __FUNC__
#define __FUNC__ "integral_fini"
HARNESS_PLUGINS_FINI(integral_fini)
{
    int           error;
    void         *instances;
    unsigned int  index;

    INTEGRAL_PRINT((
        INTEGRAL_INFO(libintegral is shutting down)))

    /* Lock plug-in plug-in mutex. */
    if (0 != (error = pthread_mutex_lock(&integral_data.mutex)))
    {
        errno = error;
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to lock integral plug-in mutex)))
        harness_syserr();
        return -1;
    }

    /* Search for handle in instances array. */
    for (index = 0; index < integral_data.count; index++)
    {
        if (integral_data.instances[index].handle == handle)
        {
            break;
        }
    }
    /* Check if handle is not in instances array. */
    if (index == integral_data.count)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(handle is not in instances array)))
        harness_syserr();
        /* Unlock harness-example plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
        {
            errno = error;
            INTEGRAL_PRINT((
                INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
            harness_syserr();
        }
        return -1;
    }

    /* Remove instance from instances array. */
    integral_data.count--;
    memmove(integral_data.instances + index,
```

```
            integral_data.instances + index +1,
            (integral_data.count   - index) *
            sizeof(unsigned int));

    /* Reallocate instances array. */
    if (0 == integral_data.count)
    {
        free(integral_data.instances);
        integral_data.instances = NULL;
    }
    else if (NULL == (instances = realloc( integral_data.instances,
                                          integral_data.count *
                                  sizeof( integral_data.instances[0]))))
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to reallocate instances array)))
    }
    else
    {
        integral_data.instances = instances;
    }
    /* Check for last finalization. */
    if (0 == integral_data.count)
    {
        /**************************/
        /* PUT YOUR FINI CODE HERE */
        /**************************/
        integral_fini_rmix();
    }

    /* Unlock integral plug-in mutex. */
    if (0 != (error = pthread_mutex_unlock(&integral_data.mutex)))
    {
        errno = error;
        INTEGRAL_PRINT((
        INTEGRAL_WARN(unable to unlock integral plug-in mutex)))
        harness_syserr();
        return -1;
    }
    return 0;
}


/*  Initializes the necessary rmix parameter and exports the plug-ins.
 *
 *  \return 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "integral_init_rmix"
int integral_init_rmix()
{
    int ret;

    RMIX_LOG((RMIX_INFO(start initialization of RMIX)))
    INTEGRAL_PRINT((
        INTEGRAL_INFO(start initialization of RMIX)))

    /* Load RMIX plug-in and initialize RMIX */

    ret = harness_plugins_load( &integral_rmix_data.handle_rmix,
                               "libharness-rmix.0", HARNESS_PLUGINS_EXPORT);
    if (ret != 0)
    {
```

```
            INTEGRAL_PRINT((
                INTEGRAL_WARN(unable to load RMIX plug-in)))
            return -1;
        }

        /* Exports Harness while forcing 1000 as specific object handle */
        ret = harness_rmix_export4( &integral_rmix_data.localref_harness,
                                    "PROTOCOL=RPC OBJECTID=1000");
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to export Harness kernel)))
            return -1;
        }

        /* Exports integral plug-in while forcing 1001 as specific object handle */
        ret = rmix_export4( &integral_rmix_data.localref_integral,
                    "PROTOCOL=RPC OBJECTID=1001", NULL, &rmixintegral_interface);
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to export integral interface)))
            return -1;
        }

        INTEGRAL_PRINT((
            INTEGRAL_INFO(RMIX initialized and integral plug-in exported)))

        return 0;
    }


    /*  Finalizes rmix and unexports the plug-ins.
     *
     *  \return 0 on success or -1 on any error
     */
    #undef  __FUNC__
    #define __FUNC__ "integral_fini_rmix"
    int integral_fini_rmix()
    {
        int ret;
        int err = 0;

        INTEGRAL_PRINT((
            INTEGRAL_INFO(start finalization of RMIX)))

        /* Unexports integral plug-in */
        ret = rmix_unexport( &integral_rmix_data.localref_integral);
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to unexport integral interface)))
            err = -1;
        }

        /* Unexports Harness */
        ret = harness_rmix_unexport( &integral_rmix_data.localref_harness);
        if (ret != 0)
        {
            RMIX_LOG((RMIX_WARN(unable to unexport Harness kernel)))
            err = -1;
        }

        /* Unload RMIX plug-in */
        ret = harness_plugins_unload( integral_rmix_data.handle_rmix);
        if (ret != 0)
```

```
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to unload RMIX plug-in)))
        err = -1;
    }

    INTEGRAL_PRINT((
        INTEGRAL_INFO(RMIX finalized and integral plug-in unexported)))

    return err;
}


/*
 * Invokes the integration (server-side method stub).
 *
 * Calculates an integral with the received data and the Monte Carlo approach.
 *
 * \param   object  The local object.
 * \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 dynamically or explicitly set to NULL by this
 *                  function. In the case of variable arrays, the length value
 *                  is allocated before the call and is set to 0 if its
 *                  variable array is NULL.
 * \param   outcnt  The output values count.
 * \param   inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 * \param   incnt   The input values count.
 * \return          0 on success or -1 on any error with errno set
 *                  appropriately.
 */
#undef  __FUNC__
#define __FUNC__ "rmixintegral_integration_call"
RMIX_METHOD_CALL(rmixintegral_integration_call)
{
    int             result;

    unsigned int    iterations;
    double          lowerbound;
    double          upperbound;
    unsigned int    numcoeffs;
    double         *coeffs;
    double          integral;

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/

    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
```

```
            errno = EINVAL;
            return -1;
        }

        /* Check outcnt parameter. */
        if ((0 == outcnt)&&(NULL != outary))
        {
            errno = EINVAL;
            RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
            errno = EINVAL;
            return -1;
        }

        /* Check inary parameter. */
        if ((NULL == inary)&&(0 != incnt))
        {
            errno = EINVAL;
            RMIX_LOG((RMIX_WARN(inary parameter is null)))
            errno = EINVAL;
            return -1;
        }

        /* Check incnt parameter. */
        if ((0 == incnt)&&(NULL != inary))
        {
            errno = EINVAL;
            RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
            errno = EINVAL;
            return -1;
        }
#endif /* DEBUG */

        /* Prepare input. */
        iterations = *(unsigned int*)inary[0];
        lowerbound = *(double*)      inary[1];
        upperbound = *(double*)      inary[2];
        numcoeffs  = *(unsigned int*)inary[3];
        coeffs     =  (double*)      inary[4];

        /* Call method function. */
        result = rmixintegral_integration(&integral, iterations, lowerbound,
                                          upperbound, numcoeffs, coeffs);

        /* Prepare output. */
        *(int*)    (outary[0]) = result;
        *(double*)(outary[1]) = integral;

        /* Reset errno if needed. */
        if (0 != result)
        {
            errno = 0;
        }
        return 0;
}


/*
 *  Calculates an integral with the received data and the Monte Carlo approach.
 *
 *  \param   *result      Returns the calculated integral
 *  \param    iterations  Number of random Monte Carlo numbers
 *  \param    lowerbound  Lower boundary of the integration interval
 *  \param    upperbound  Upper boundary of the integration interval
```

```
 *  \param    numcoeffs   Number of coefficients
 *  \param   *coeffs      Array with coefficients
 *  \return               0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmixintegral_integration"
int rmixintegral_integration ( double        *result,
                               unsigned int   iterations,
                               double         lowerbound,
                               double         upperbound,
                               unsigned int   numcoeffs,
                               double        *coeffs)
{
    int i;
#ifdef DEBUG
    /* check array parameter */
    if (coeffs == NULL)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(array of coefficients contains no data)))
        return -1;
    }
#endif /* DEBUG */

    /* initialise the random generator */
    srand(time(NULL));

    /* computation */
    for ( i=0; i<iterations; i++)
    {
        (*result) += integral_func_value( numcoeffs, coeffs,
                        integral_random_number( lowerbound, upperbound));
    }
    (*result) = (*result) * (upperbound - lowerbound) / iterations;

    /* for testing purposes, the plug-in was slowed so that it can be shutdown
       manually */
    /*
    RMIX_LOG((RMIX_INFO(------------------------------------- )))
    RMIX_LOG((RMIX_INFO(beende plugin  lege plugin schlafen )))
    RMIX_LOG((RMIX_INFO(------------------------------------ )))
    sleep(30);
    */

    INTEGRAL_PRINT((
        INTEGRAL_INFO(integral is %lf), (*result)))

    return 0;
}


/*
 *  Calculates the function value of a given function.
 *
 *  \param   num_coeffs  Number of coefficients of the functions.
 *  \param  *coeffs      Array with the coefficients
 *  \param   x_value     Value, which is inserted into the function
 *  \return              The function value.
 */
#undef  __FUNC__
#define __FUNC__ "integral_func_value"
double integral_func_value( unsigned int  num_coeffs,
                            double       *coeffs,
```

```
                                double         x_value)
{
    int i;
    double erg = 0;
    for ( i=0; i<num_coeffs; i++)
    {
        erg += coeffs[i] * pow(x_value, (double) i);
    }
    return erg;
}


/*
 *  Generates a random number in a certain interval.
 *
 *  \param  bottom  Lower boundary of the interval.
 *  \param  top     Upper boundary of the interval
 *  \return         The random number
 */
#undef  __FUNC__
#define __FUNC__ "integral_random_number"
double integral_random_number( double bottom,
                               double top)
{
    double random = 0;
    random = (double)rand() / RAND_MAX;
    random = random * (top - bottom) + bottom;
    return random;
}


/*****************************************************************************
 *
 *  The following section contains for testing the plug-in without the use
 *  of communication functions. A local input file is read and a integral
 *  is calculated.
 *
*****************************************************************************/

/*
 *  Calculates the integral of a given function in a certain interval.
 *
 *  \return 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "integral_calc_integral"
int integral_calc_integral()
{
    char *filename;

    unsigned int num_of_coeffs;    /* number of coefficients */
    int iterations;                /* number of iterations */
    double *coeffs;                /* array for storing the coefficients */

    double bottom_border;          /* bottom border for the calculation area */
    double top_border;             /* top border of the calculation area */

    double result = 0;             /* atores the result */
    int i, ret;

    /* example file for testing the plug-in without a loader */
    filename = TESTFILE;

```

```
    /* read the input file */
    ret = integral_read_inputfile( filename, &num_of_coeffs, &coeffs,
                                   &iterations, &bottom_border, &top_border);
    if ( ret == -1 )
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to read input file %s), filename))
        return -1;
    }

    /* initialise the random generator */
    srand(time(NULL));

    /* computation */
    for ( i=0; i<iterations; i++)
    {
        result += integral_func_value( num_of_coeffs, coeffs,
                          integral_random_number( bottom_border, top_border));
    }
    result = result * (top_border - bottom_border) / iterations;

    INTEGRAL_PRINT((
        INTEGRAL_INFO(integral is %lf), result))

    FREE(coeffs);
    return 0;
}


/*
 *  Reads the input file, which contains the needed information.
 *
 *  \param  *filename   Name of the input file
 *  \param  *number     Returns the number of coefficients
 *  \param  **coeffs    Returns the array of coefficients
 *  \param  *iterations Returns the amount of random numbers, which are used
 *  \param  *bottom     Returns the lower boundary of the interval
 *  \param  *top        Returns the upper boundary of the interval
 *  \return             0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "integral_read_inputfile"
int integral_read_inputfile( char           *filename,
                             unsigned int   *number,
                             double        **coeffs,
                             int            *iterations,
                             double         *bottom,
                             double         *top)
{
    FILE *dat_file;     /* input file */
    int i;
    int ret;


    /* open the file with the coefficients */
    dat_file = fopen(filename, "r");
    if (dat_file == NULL)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to open config file)))
        return -1;
    }
    else
```

```
    {
        INTEGRAL_PRINT((
            INTEGRAL_INFO(input file is opened)))
    }

    /* read in the number of iterations */
    ret = fscanf(dat_file, "%d", iterations);
    if (ret == 0 || ret == -1)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to read iterations from file)))
        fclose(dat_file);
        return -1;
    }

    /* read in the boundaries */
    ret = fscanf(dat_file, "%lf", bottom);
    if (ret == 0 || ret == -1)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to read lower boundary from file)))
        fclose(dat_file);
        return -1;
    }
    ret = fscanf(dat_file, "%lf", top);
    if (ret == 0 || ret == -1)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to read upper boundary from file)))
        fclose(dat_file);
        return -1;
    }

    /* read in the number of coefficients */
    ret = fscanf(dat_file, "%d", number);
    if (ret == 0 || ret == -1)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to read number of coefficients from
                                    file)))
        fclose(dat_file);
        return -1;
    }

    /* create the array */
    (*coeffs) = (double *)malloc( sizeof(double) * (*number));
    if ((*coeffs) == NULL)
    {
        INTEGRAL_PRINT((
            INTEGRAL_WARN(unable to allocate memory for coefficients)))
        fclose(dat_file);
        return -1;
    }

    /* read in the coefficients */
    for (i=0; i<(*number); i++)
    {
        ret = fscanf(dat_file, "%lf", &(*coeffs)[i]);
        if (ret == 0 || ret == -1)
        {
            INTEGRAL_PRINT((
                INTEGRAL_WARN(unable to read coefficients from file)))
            FREE(coeffs);
```

```
            fclose(dat_file);
            return -1;
        }
    }

    fclose(dat_file);
    return 0;
}

/*****************************************************************************
 *
 * END OF FILE
 *
 *****************************************************************************/
```

A.3.4. Parallel Plug-in for a Image Processing Pipeline

A.3.4.1. Header File

```
/* libimgproc/imgproc.h.  Generated by configure.  */
/*****************************************************************************
 *
 * Header file for the image processing module.
 * Copyright (c) Ronald Baumann
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 * Process the '.in' file with `configure` or `autogen.sh` from the distribution
 * top-level directory to create the target file.
 *
 *****************************************************************************/

/** \file imgproc.h
 *  \brief Header file for image processing module.
 *
 *  The libimgproc module is a plug-in for the Harness project. It performes
 *  various image filters on pictures.
 */

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


/*****************************************************************************
 *
 * Macros
 *
 *****************************************************************************/

/** \def    LIBIMGPROC_PACKAGE_NAME
 *  \brief Package name (unquoted).
 */
```

```
#define LIBIMGPROC_PACKAGE_NAME imageprocessing

/** \def    LIBIMGPROC_PACKAGE_VERSION
 *  \brief Package version (unquoted).
 */
#define LIBIMGPROC_PACKAGE_VERSION 1.1

/** \def    LIBIMGPROC_PACKAGE_RELEASE
 *  \brief Package release (unquoted).
 */
#define LIBIMGPROC_PACKAGE_RELEASE 0

/** \def    LIBIMGPROC_PACKAGE_BUGREPORT
 *  \brief Package bug report e-mail (unquoted).
 */
#define LIBIMGPROC_PACKAGE_BUGREPORT baumannr@ornl.gov

/** \def    LIBIMGPROC_VERSION_CURRENT
 *  \brief Version current (unquoted).
 */
#define LIBIMGPROC_VERSION_CURRENT 0

/** \def    LIBIMGPROC_VERSION_REVISION
 *  \brief Version revision (unquoted).
 */
#define LIBIMGPROC_VERSION_REVISION 0

/** \def    LIBIMGPROC_VERSION_AGE
 *  \brief Version age (unquoted).
 */
#define LIBIMGPROC_VERSION_AGE 0

/** \def    LIBIMGPROC_VERSION_FIRST
 *  \brief Version first (unquoted).
 */
#define LIBIMGPROC_VERSION_FIRST 0

/** \def    LIBIMGPROC_VERSION
 *  \brief Version (unquoted).
 */
#define LIBIMGPROC_VERSION 0.0.0

/** \def    IMGPROC_QUOTES(string)
 *  \brief Quoting a string.
 */
#define IMGPROC_QUOTES(string) #string

/** \def    IMGPROC_STRING(string)
 *  \brief A string.
 */
#define IMGPROC_STRING(string) IMGPROC_QUOTES(string)

```

```
/** \def    IMGPROC_FUSE(arg1, arg2)
 *  \brief Fuse two strings.
 */
#define IMGPROC_FUSE(arg1, arg2) arg1##arg2


/** \def    IMGPROC_JOIN(arg1, arg2)
 *  \brief Joining two text constants.
 */
#define IMGPROC_JOIN(arg1, arg2) IMGPROC_FUSE(arg1, arg2)


/** \def    IMGPROC_WARN(string)
 *  \brief Debug printout.
 */
#define IMGPROC_WARN(string) \
  fprintf(stderr,\
          IMGPROC_STRING(warn:libimgproc:%d:%s:%u:%s:string\n),\
          getpid(), __FILE__, __LINE__, __FUNC__


/** \def    IMGPROC_INFO(string)
 *  \brief Debug printout.
 */
#define IMGPROC_INFO(string) \
  fprintf(stderr,\
          IMGPROC_STRING(info:libimgproc:%d:%s:%u:%s:string\n),\
          getpid(), __FILE__, __LINE__, __FUNC__


/** \def    IMGPROC_PRINT(string)
 *  \brief Wrapper for debug printout.
 */
#ifdef      DEBUG
#define     IMGPROC_PRINT(string) string);
#else   /* DEBUG */
#define     IMGPROC_PRINT(string)
#endif /* DEBUG */


/* Flag for <harness.0/harness.h> header. */
#ifndef      HARNESS_HARNESS_H
#ifndef      HAVE_HARNESS_0_HARNESS_H
#define       HAVE_HARNESS_0_HARNESS_H 1
#endif   /* HAVE_HARNESS_0_HARNESS_H */
#endif   /* HARNESS_HARNESS_H */


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


/** \def ThrowWandException(wand)
 *  \brief Prints out error information caused by Magick Wand library.
 */
#define ThrowWandException(wand) \
    { \
        char *description; \
        ExceptionType severity; \
```

```
        description = MagickGetException( wand, &severity ); \
        fprintf( stderr, "%s %s %ld %s\n", GetMagickModule(), description); \
        description = (char *) MagickRelinquishMemory(description); \
    }


/** \def FILTER_NUMBER
 *  \brief Number of image filters.
 */
#define FILTER_NUMBER 5


/************************************************************************
 *
 * Includes
 *
 ************************************************************************/

/* Include <harness.0/harness.h> header. */
#if        HAVE_HARNESS_0_HARNESS_H
#include       <harness.0/harness.h>
#endif /* HAVE_HARNESS_0_HARNESS_H */

#include <wand/magick_wand.h>
#include <dirent.h>
#include "rmiximgproc.h"
#include "rmixppm.h"


/************************************************************************
 *
 * Data Types
 *
 ************************************************************************/

/** \struct imgproc_t
 *  \brief Integral plug-in "class".
 *
 *  This structure contains the public plug-in data and function pointers.
 *  It also provides information about the version of the plug-in.
 */
typedef struct
{
    struct
    {
        const unsigned int current;     /* current version         */
        const unsigned int revision;    /* current revision        */
        const unsigned int age;         /* version age             */
        const unsigned int first;       /* first supported version */
    }version;                           /* plug-in version         */
}imgproc_t;


/** \struct worklist_t
 *  \brief  Element of the linked list for storing iamge information.
 *
 *  Each element contains name, size and the image data from images, which were
 *  processed by the plug-in.
 */
typedef struct worklist
{
    char              *name;     /* name of the image                */
    size_t             size;     /* size of the image                */
```

```
    unsigned char    *blob;      /* image data                    */
    struct   worklist *next;     /* pointer to the next list element */
}worklist_t;


/***************************************************************************
 *
 * Function Prototypes
 *
 ***************************************************************************/

/** \fn      int imgproc_init_rmix();
 *  \brief   Initializes the necessary rmix parameter and exports the plug-in.
 *
 *  \return 0 on success or -1 on any error
 */
int imgproc_init_rmix();


/*  \fn      int imgproc_fini_rmix();
 *  \brief   Finalizes rmix and unexports the plug-in.
 *
 *  \return 0 on success or -1 on any error
 */
int imgproc_fini_rmix();


/** \fn      RMIX_METHOD_CALL(rmiximgproc_initpipeline_call);
 *  \brief   Accepts calls for the initialisation of the pipeline component
 *           (server-side method stub).
 *
 *  \param   object  The local object.
 *  \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
 *                   dynamically or explicitly set to NULL by this function. In
 *                   the case of variable arrays, the length value is allocated
 *                   before the call and is set to 0 if its variable array is
 *                   NULL.
 *  \param   outcnt  The output values count.
 *  \param   inary   The input values array. The values array and its containing
 *                   values are not modified or deallocated by this call.
 *  \param   incnt   The input values count.
 *  \return          0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_initpipeline_call);


/** \fn      int rmiximgproc_initpipeline (unsigned int   filter, \
                                           char           *sourcedir, \
                                           char           *targetdir, \
                                           char           *successor \
                                           char           *predecessor \
                                           char           *ppmnode);
 *  \bief    Initialises the global pipeline parameters.
 *
 *  \param    filter         Filter which will be used
 *  \param   *sourcedir      Source images dir
 *  \param   *targetdir      Target dir for images
 *  \param   *sucessor       Successor plug-in
 *  \param   *predecessor    Predecessor plug-in
 *  \param   *ppmnode        Node running the PPM
 *  \return                  0 on success or -1 on any error.
```

```
 */
int rmiximgproc_initpipeline (unsigned int   filter,
                              char          *sourcedir,
                              char          *targetdir,
                              char          *successor,
                              char          *predecessor,
                              char          *ppmnode);

/** \fn     RMIX_METHOD_CALL(rmiximgproc_invokepipeline_call);
 *  \brief  Accepts invocation calls for the pipeline (server-side method stub).
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_invokepipeline_call);

/** \fn     int rmiximgproc_invokepipeline ();
 *  \brief  Starts the image processing pipeline.
 *
 *  \return 0 on success or -1 on any error.
 */
int rmiximgproc_invokepipeline ();

/** \fn     RMIX_METHOD_CALL(rmiximgproc_availabilitycheck_call);
 *  \brief  Accepts checking calls, if the plug-in was loaded correctly
 *          (server-side method stub).
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_availabilitycheck_call);

/** \fn     RMIX_METHOD_CALL(rmiximgproc_setimagecounter_call);
 *  \brief  Accepts initialisation calls for the image counter of the plug-in
 *          (server-side method stub).
 *
 *  \param  object  The local object.
```

```
 *  \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
 *                   dynamically or explicitly set to NULL by this function. In
 *                   the case of variable arrays, the length value is allocated
 *                   before the call and is set to 0 if its variable array is
 *                   NULL.
 *  \param   outcnt  The output values count.
 *  \param   inary   The input values array. The values array and its containing
 *                   values are not modified or deallocated by this call.
 *  \param   incnt   The input values count.
 *  \return          0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_setimagecounter_call);


/** \fn      int rmiximgproc_setimagecounter( unsigned int counter);
 *  \brief   Sets the image counter of the plug-in.
 *
 *  \param    counter    Number of images to process
 *  \return              0 on success or -1 on any error.
 */
int rmiximgproc_setimagecounter( unsigned int counter);


/** /fn      int rmiximgprocclient_setimagecounter( rmix_remoteref_t *remoteobj,\
                                                    unsigned int      counter);
 *  \brief   Calls the function for setting an image counter at another plug-in
 *           (client-side method stub).
 *
 *  \param    *remoteobj  ID of the remote object
 *  \param     counter    Image counter
 *  \return               0 on success or -1 on any error
 */
int rmiximgprocclient_setimagecounter ( rmix_remoteref_t *remoteobj,
                                        unsigned int      counter);


/** \fn      RMIX_METHOD_CALL(rmiximgproc_passimage_call);
 *  \brief   Accepts callsm, which forward an image to the plug-in
 *           (server-side method stub).
 *
 *  \param   object  The local object.
 *  \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
 *                   dynamically or explicitly set to NULL by this function. In
 *                   the case of variable arrays, the length value is allocated
 *                   before the call and is set to 0 if its variable array is
 *                   NULL.
 *  \param   outcnt  The output values count.
 *  \param   inary   The input values array. The values array and its containing
 *                   values are not modified or deallocated by this call.
 *  \param   incnt   The input values count.
 *  \return          0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_passimage_call);


/** \fn      int rmiximgproc_passimage (char          *name, \
                                       size_t         size, \
                                       unsigned char *blob);
 *  \brief   Processes and then forwards an image to the next plug-in.
```

```
 *
 * \param    size    Size of image data
 * \param   *blob    Image data
 * \return           0 on success or -1 on any error.
 */
int rmiximgproc_passimage (char           *name,
                           size_t          size,
                           unsigned char *blob);


/** \fn     int rmiximgprocclient_passimage_oneway (
                                  rmix_remoteref_t *remoteobj, \
                                  char             *name,\
                                  unsigned char    *blob,\
                                  unsigned int      size);
 * \brief  Calls a plug-in for forwarding an image (client-side method stub).
 *
 * \param  *name   Name of the image file
 * \param  *blob   Image data
 * \param   size   Size of the image data
 * \return         0 on success or -1 on any error
 */
int rmiximgprocclient_passimage_oneway ( rmix_remoteref_t *remoteobj,
                                         char             *name,
                                         unsigned char    *blob,
                                         unsigned int      size);


/** \fn     RMIX_METHOD_CALL(rmiximgproc_updatesuccessor_call);
 * \brief  Accepts calls for updating the predecessor entry of the plug-in.
 *         (server-side stub)
 *
 * \param  object  The local object.
 * \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
 *                 dynamically or explicitly set to NULL by this function. In
 *                 the case of variable arrays, the length value is allocated
 *                 before the call and is set to 0 if its variable array is
 *                 NULL.
 * \param  outcnt  The output values count.
 * \param  inary   The input values array. The values array and its containing
 *                 values are not modified or deallocated by this call.
 * \param  incnt   The input values count.
 * \return         0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_updatesuccessor_call);


/** \fn     int rmiximgproc_updatesuccessor (char *predecessor);
 * \brief  Updates the predecessor entry of the plug-in.
 *
 * \param  *predecessor  Name of the predecessor plug-in.
 * \return               0 on success or -1 on any error.
 */
int rmiximgproc_updatesuccessor (char *predecessor);


/** \fn     RMIX_METHOD_CALL(rmiximgproc_updatepredecessor_call);
 * \brief  Accepts calls for updating the successor entry of the plug-in and
 *         also returns the current image counter. (server-side stub)
 *
 * \param  object  The local object.
```

```
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_updatepredecessor_call);


/** \fn     int rmiximgproc_updatepredecessor( char         *successor, \
                                              unsigned int *counter);
 *  \brief  Updates the successor entry of the plug-in and returns the current
 *          image counter.
 *
 *  \param   *successor  Name of the successor plug-in.
 *  \param   *counter    Current image counter.
 *  \return              0 on success or -1 on any error.
 */
int rmiximgproc_updatepredecessor( char         *successor,
                                   unsigned int *counter);


/** \fn     RMIX_METHOD_CALL(rmiximgproc_updateimagecounter_call);
 *  \brief  Accepts calls for updating the image counter. (server-side stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_updateimagecounter_call);


/** \fn     int rmiximgproc_updateimagecounter( unsigned int counter);
 *  \brief  Updates the image counter.
 *
 *  \param   counter   Number of images to process
 *  \return            0 on success or -1 on any error.
 */
int rmiximgproc_updateimagecounter( unsigned int counter);


/** \fn     RMIX_METHOD_CALL(rmiximgproc_sendworklist_call);
 *  \brief  Accepts call for sending the images in the work list again to the
 *          successor plug-in (server-side stub).
 *
 *  \param  object  The local object.
```

```
 *  \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
 *                    dynamically or explicitly set to NULL by this function. In
 *                    the case of variable arrays, the length value is allocated
 *                    before the call and is set to 0 if its variable array is
 *                    NULL.
 *  \param   outcnt   The output values count.
 *  \param   inary    The input values array. The values array and its containing
 *                    values are not modified or deallocated by this call.
 *  \param   incnt    The input values count.
 *  \return           0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_sendworklist_call);


/** \fn      int rmiximgproc_sendworklist();
 *  \brief   Sends again the images in the worklist to the successor plug-in.
 *
 *  \return 0 on success or -1 on any error.
 */
int rmiximgproc_sendworklist();


/** \fn      RMIX_METHOD_CALL(rmiximgproc_imageprocessed_call);
 *  \brief   Accepts notification calls that an image was successfully stored
 *           (server-side stub).
 *
 *  \param   object   The local object.
 *  \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
 *                    dynamically or explicitly set to NULL by this function. In
 *                    the case of variable arrays, the length value is allocated
 *                    before the call and is set to 0 if its variable array is
 *                    NULL.
 *  \param   outcnt   The output values count.
 *  \param   inary    The input values array. The values array and its containing
 *                    values are not modified or deallocated by this call.
 *  \param   incnt    The input values count.
 *  \return           0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_imageprocessed_call);


/** \fn      int rmiximgproc_imageprocessed (char* name);
 *  \brief   Deletes a processed image from the internal worklist. If all images
 *           are processed the plug-in is shutdown.
 *
 *  \param    *name   Name of the image.
 *  \return           0 on success or -1 on any error.
 */
int rmiximgproc_imageprocessed (char* name);


/** \fn      int rmiximgprocclient_imageprocessed_oneway ( \
                                            rmix_remoteref_t *remoteobj,\
                                            char              *name);
 *  \brief   Calls a plug-in for informing it that an image was
 *           successfully stored (client-side method stub).
 *
 *  \param   *remoteobj  ID of the remote object
 *  \param   *name       Name of the image file
```

```
 *  \return             0 on success or -1 on any error
 */
int rmiximgprocclient_imageprocessed_oneway ( rmix_remoteref_t *remoteobj,
                                              char             *name);


/** \fn     int rmixppmclient_repairpipe ( rmix_remoteref_t *remoteobj,\
                                          char             *node);
 *  \brief  Calls the Parallel Plug-in Manager for repairing the pipeline
 *          (client-side method stub).
 *
 *  \param  *remoteobj  ID of the remote object
 *  \param  *node       Name of the not reachable node
 *  \return             0 on success or -1 on any error
 */
int rmixppmclient_repairpipe ( rmix_remoteref_t *remoteobj,
                               char             *node);


/** \fn     int imgproc_createremoteref( rmix_remoteref_t **remoteobj,\
                                        char              *objectid,\
                                        char              *node);
 *  \brief  Creates a remote reference specified by objectid and node. The RPC
 *          protocol is used.
 *
 *  \param    **remoteobj  ID of the remote object
 *  \param     *objectid   ID of the exported object
 *  \param     *node       Name of the node
 *  \return                0 on success or -1 on any error
 */
int imgproc_createremoteref( rmix_remoteref_t **remoteobj,
                             char              *objectid,
                             char              *node);


/*****************************************************************************
 *
 *  The following section contains the image processing pipeline functions.
 *
 *****************************************************************************/

/** \fn     int imgproc_decrease_imagecounter();
 *  \brief  Decreases the image counter by 1.
 *
 *  \return 0 on success or -1 on any error
 */
int imgproc_decrease_imagecounter();


/** \fn     int imgproc_create_magickwand( char *imgfile, \
                                          char *imgname);
 *  \brief  Processes a loaded image on the first plug-in unit of the pipeline
 *          and forwards it to the next unit.
 *
 *  \param  *imgfile  Image namne + path
 *  \param  *imgname  Image name
 *  \return           0 on success or -1 on any error
 */
int imgproc_create_magickwand( char *imgfile,
                               char *imgname);


/** \fn     int imgproc_filter_rotation( unsigned char **blob, \
```

```
                                              size_t         *size);
 *  \brief   Rotates an image by 90 degrees.
 *
 *  \param   **blob   Char array containing the image data.
 *  \param   *size    Size of the char array.
 *  \return           0 on success or -1 on any error
 */
int imgproc_filter_rotation( unsigned char **blob,
                             size_t         *size);

/** \fn      int imgproc_filter_oilpainting( unsigned char **blob, \
                                              size_t         *size);
 *  \brief   Puts an oil painting filter on an image.
 *
 *  \param   **blob   Char array containing the image data.
 *  \param   *size    Size of the char array.
 *  \return           0 on success or -1 on any error
 */
int imgproc_filter_oilpainting( unsigned char **blob,
                                size_t         *size);

/** \fn      int imgproc_filter_swirl( unsigned char **blob, \
                                        size_t         *size);
 *  \brief   Swirls the center of an image.
 *
 *  \param   **blob   Char array containing the image data.
 *  \param   *size    Size of the char array.
 *  \return           0 on success or -1 on any error
 */
int imgproc_filter_swirl( unsigned char **blob,
                          size_t         *size);

/** \fn      int imgproc_filter_negate( unsigned char **blob, \
                                         size_t         *size);
 *  \brief   Negates an image.
 *
 *  \param   **blob   Char array containing the image data.
 *  \param   *size    Size of the char array.
 *  \return           0 on success or -1 on any error
 */
int imgproc_filter_negate( unsigned char **blob,
                           size_t         *size);

/** \fn      int imgproc_filter_solarize( unsigned char **blob, \
                                           size_t         *size);
 *  \brief   Puts a solarize filter on an image.
 *
 *  \param   **blob   Char array containing the image data.
 *  \param   *size    Size of the char array.
 *  \return           0 on success or -1 on any error
 */
int imgproc_filter_solarize( unsigned char **blob,
                             size_t         *size);

/** \fn      int imgproc_write_image( unsigned char *blob, \
                                       size_t         size, \
                                       char          *filename);
 *  \brief   Stores an image.
```

```
 *
 *  \param   **blob      Char array containing the image data.
 *  \param    *size      Size of the char array.
 *  \param    *filename  Name of the new image file.
 *  \return              0 on success or -1 on any error
 */
int imgproc_write_image( unsigned char *blob,
                         size_t         size,
                         char          *filename);


/** \fn     int imgproc_get_files( char *dir);
 *  \brief  Returns the number of files found in a directory.
 *
 *  \param   *dir     Directory name.
 *  \return           Number of files.
 */
int imgproc_get_files( char *dir);


/** \fn     int imgproc_loadimages( char  *imgdir);
 *  \brief  Tries to load the image files in the specified directory and
 *          additional files.
 *
 *  \param    *imgdir      Directory containing the image files.
 *  \return                0 on success or -1 on any error
 */
int imgproc_loadimages( char  *imgdir);


/*****************************************************************************
 *
 * next section contains functions for accessing the image worklist
 *
 *****************************************************************************/

/** \fn     int imgproc_worklist_insert( char           *name,\
                                         size_t          size,\
                                         unsigned char  *blob, \
                                         worklist_t    **list_pointer);
 *  \brief  Inserts an element in the worklist.
 *
 *  \param    *name           Name of the image
 *  \param     size           Size of the image
 *  \param    *blob           Data of the image
 *  \param   **list_pointer   Pointer to the linked list
 *  \return                   0 on success or -1 on any error
 */
int imgproc_worklist_insert( char           *name,
                             size_t          size,
                             unsigned char  *blob,
                             worklist_t    **list_pointer);


/** \fn     void imgproc_worklist_destroylist( worklist_t **list_pointer);
 *  \brief  Deletes the linked list defined by the pointer.
 *
 *  \param   **list_pointer Pointer to the linked list
 */
void imgproc_worklist_destroylist( worklist_t **list_pointer);


/** \fn     int imgproc_worklist_delete( char        *name, \
```

```
                                            worklist_t **list_pointer);
 *  \brief  Deletes the element of the list defined by the name.
 *
 *  \param   *name          Name of the element.
 *  \param  **list_pointer Pointer to the linked list
 *  \return                 0 on success otherwise -1
 */
int imgproc_worklist_delete( char        *name,
                             worklist_t **list_pointer);


/** \fn     void imgproc_worklist_printlist( worklist_t *list_pointer);
 *  \brief  Prints the names of the list elements.
 *
 *  \param  *list_pointer Pointer to the linked list
 */
void imgproc_worklist_printlist( worklist_t *list_pointer);


/** \fn     int imgproc_worklist_find_elementposition(char       *entry, \
                                                      worklist_t *list_pointer);
 *  \brief  Finds the position of an entry.
 *
 *  \param  *char         Entry to find
 *  \param  *list_pointer Pointer to the linked list
 *  \return               The position or otherwise -1
 */
int imgproc_worklist_find_elementposition( char       *entry,
                                           worklist_t *list_pointer);


/******************************************************************************
 *
 * Data
 *
 ******************************************************************************/

/** \var extern imgproc_t imgproc
 *  \brief Integral plug-in "object".
 *
 *  An external "object" of the Integral plug-in "class".
 */
extern imgproc_t imgproc;


#endif /* IMGPROC_IMGPROC_H */

/******************************************************************************
 *
 * END OF FILE
 *
 ******************************************************************************/
```

A.3.4.2. Source File

```
/******************************************************************************
 *
 * Source file for the image processing module.
 * Copyright (c) Ronald Baumann.
 *
 * For more information see the following files in the source distribution top-
```

```
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 ***************************************************************************/

/** \file  imgproc.c
 *  \brief Sorce file for image processing module.
 *
 *  The libimgproc module is a plug-in for the Harness project. It performes
 *  various image filters on pictures.
 */


/***************************************************************************
 *
 * Includes
 *
 ***************************************************************************/

/* Main module header file. */
#include "imgproc.h"


/***************************************************************************
 *
 * Data Types
 *
 ***************************************************************************/

/** \struct imgproc_data_t
 *  \brief  Image processing plug-in module data type.
 *
 *  Contains the mutex and the instances handler.
 */
typedef struct
{
    pthread_mutex_t  mutex;         /* mutex                      */
    unsigned int     count;         /* instances array count      */
    struct
    {
        unsigned int handle;        /* instance handle            */
    }*instances;                    /* instances array            */

    unsigned int   filter;          /* filter which will be used  */
    char          *sourcedir;       /* source images dir          */
    char          *targetdir;       /* target dir for images      */
    char          *successor;       /* successor plug-in          */
    char          *predecessor;     /* predecessor plug-in        */
    char          *ppmnode;         /* node running the PPM       */
    unsigned int   imagecounter;    /* counter for images to process */
}imgproc_data_t;


/** \struct imgproc_rmix_data_t
 *  \brief  Data for rmix communication functions.
 */
typedef struct
{
```

```
    rmix_localref_t *localref_harness;     /* exported Harness kernel   */
    rmix_localref_t *localref_imgproc;     /* exported imgproc object   */
    unsigned int      handle_rmix;         /* handle for rmix plug-in   */
}imgproc_rmix_data_t;


/** \var    int(*pFunc)( unsigned char **, size_t *);
 *  \brief  Definition of a pointer to a image filter function.
 */
typedef int(*pFunc)( unsigned char **, size_t *);


/***************************************************************************
 *
 * Function Prototypes
 *
 ***************************************************************************/

/** \fn     HARNESS_PLUGINS_INIT(imgproc_init);
 *  \brief  Initializes the image processing plug-in.
 *
 *  \param  handle The plug-in instance handle.
 *  \return 0 on success or -1 on any error with errno set appropriately.
 */
HARNESS_PLUGINS_INIT(imgproc_init);


/** \fn     HARNESS_PLUGINS_FINI(imgproc_fini);
 *  \brief  Finalizes the image processing plug-in.
 *
 *  \param  handle The plug-in instance handle.
 *  \return 0 on success or -1 on any error with errno set appropriately.
 */
HARNESS_PLUGINS_FINI(imgproc_fini);


/***************************************************************************
 *
 * Data
 *
 ***************************************************************************/

/** \var   rmix_method_t rmixppmclient_methods[RMIXPPM_METHODS_COUNT]
 *  \brief Client-side method descriptors for rmix ppm.
 */
const rmix_method_t rmixppmclient_methods[RMIXPPM_METHODS_COUNT] =
    RMIXPPM_METHODS_CLIENT;


/** \var   rmix_interface_t rmixppmclient_interface
 *  \brief Client-side interface for rmix ppm.
 */
const rmix_interface_t rmixppmclient_interface = {
    RMIXPPM_METHODS_COUNT,                 /* method descriptor count */
    (rmix_method_t*)rmixppmclient_methods  /* method descriptor array */
};


/** \var   const rmix_method_t rmiximgproc_methods[RMIXIMGPROC_METHODS_COUNT]
 *  \brief Server-side method descriptors for rmix imgproc.
 */
const rmix_method_t rmiximgproc_methods[RMIXIMGPROC_METHODS_COUNT] =
    RMIXIMGPROC_METHODS;
```

```
/** \var    const rmix_interface_t rmiximgprog_interface
 *  \brief Server-side interface for rmix imgproc.
 */
const rmix_interface_t rmiximgproc_interface =
{
    RMIXIMGPROC_METHODS_COUNT,              /* method descriptor count */
    (rmix_method_t*)rmiximgproc_methods   /* method descriptor array */
};


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


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


/** \var    pFunc filters
 *  \brief Array with pointers to the image filter functions.
 */
const pFunc filters[FILTER_NUMBER] =
{
    &imgproc_filter_rotation,
    &imgproc_filter_oilpainting,
    &imgproc_filter_swirl,
    &imgproc_filter_negate,
    &imgproc_filter_solarize
};


/** \var    imgproc_rmix_data_t imgproc_rmix_data
 *  \brief Data for rmix communication functions.
 *
 *  Includes the local references to the exported objects.
 */
imgproc_rmix_data_t imgproc_rmix_data =
{
    NULL,         /* exported Harness kernel  */
    NULL,         /* exported imgproc object  */
    0             /* handle for rmix plug-in  */
};


/** \var    imgproc_data_t imgproc_data
 *  \brief image processing plug-in module data.
 *
 *  Includes the mutex for instances handle and the instances array where the
 *  handles are stored.
 */
imgproc_data_t imgproc_data =
{
    PTHREAD_MUTEX_INITIALIZER,  /* mutex                    */
    0,                          /* instances array count    */
```

```
    NULL,                   /* instances array          */
    0,                      /* filter which will be used */
    NULL,                   /* source images dir        */
    NULL,                   /* target dir for images    */
    NULL,                   /* successor plug-in        */
    NULL,                   /* predecessor plug-in      */
    0                       /* images to process        */
};

/** \var   worklist_t worklist
 *  \brief list with image information of the processed images.
 */
worklist_t *worklist = NULL;

unsigned int testswitch = 0;

/** \var   imgproc_t imgproc
 *  \brief Image processing plug-in "object".
 *
 *  Contains the version of the library and possible public data and function
 *  pointers.
 */
imgproc_t imgproc =
{
    {
        LIBIMGPROC_VERSION_CURRENT,    /* current version         */
        LIBIMGPROC_VERSION_REVISION,   /* current revision        */
        LIBIMGPROC_VERSION_AGE,        /* version age             */
        LIBIMGPROC_VERSION_FIRST       /* first supported version */
    }                                  /* plug-in version         */
};

/****************************************************************************
 *
 * Functions
 *
 ****************************************************************************/

/*
 * Initializes the image processing plug-in plug-in.
 *
 * handle = The plug-in instance handle.
 * return = 0 on success or -1 on any error with errno set appropriately.
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_init"
HARNESS_PLUGINS_INIT(imgproc_init)
{
    int           error;
    void          *instances;
    unsigned int  index;

    IMGPROC_PRINT((
        IMGPROC_INFO(libimgproc is starting)))

    /* Lock image processing plug-in mutex. */
    if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex)))
    {
        errno = error;
```

```
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to lock image processing plug-in mutex)))
        harness_syserr();
        return -1;
    }

    /* Search for handle in instances array. */
    for (index = 0; index < imgproc_data.count; index++)
    {
        if (imgproc_data.instances[index].handle == handle)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(handle is already in instances array)))
            harness_syserr();

            /* Unlock image processing plug-in mutex. */
            if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
            {
                errno = error;
                IMGPROC_PRINT((
                    IMGPROC_WARN(unable to unlock image processing plug-in
                                 mutex)))
                harness_syserr();
            }

            return -1;
        }
    }

    /* Increase instances array. */
    index = imgproc_data.count;
    imgproc_data.count++;
    /* Reallocate instances array. */
    if (NULL == (instances = realloc( imgproc_data.instances,
                                      imgproc_data.count *
                              sizeof( imgproc_data.instances[0]))))
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to reallocate instances array)))
        /* Unlock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
            harness_syserr();
        }
        return -1;
    }

    imgproc_data.instances = instances;
    /* Save instances array entry. */
    imgproc_data.instances[index].handle = handle;
    /* Check for first initialization. */
    if (1 == imgproc_data.count)
    {
        /***************************/
        /* PUT YOUR INIT CODE HERE */
        /***************************/
        if (imgproc_init_rmix() != 0)
        {
            /* if the communication interface cannot be exported terminate
               the plug-in */
```

```
                if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
                {
                    errno = error;
                    IMGPROC_PRINT((
                        IMGPROC_WARN(unable to unlock image processing plug-in
                                     mutex)))
                    harness_syserr();
                    return -1;
                }
                harness_kernel_shutdown();
            }
        }

        /* Unlock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }
        return 0;
    }


    /*
     * Finalizes the image processing plug-in.
     *
     * handle = The plug-in instance handle.
     * return = 0 on success or -1 on any error with errno set appropriately.
     */
    #undef  __FUNC__
    #define __FUNC__ "imgproc_fini"
    HARNESS_PLUGINS_FINI(imgproc_fini)
    {
        int             error;
        void           *instances;
        unsigned int    index;

        IMGPROC_PRINT((
            IMGPROC_INFO(libimgproc is shutting down)))

        /* Lock plug-in mutex. */
        if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to lock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        /* Search for handle in instances array. */
        for (index = 0; index < imgproc_data.count; index++)
        {
            if (imgproc_data.instances[index].handle == handle)
            {
                break;
            }
        }
        /* Check if handle is not in instances array. */
        if (index == imgproc_data.count)
```

```
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(handle is not in instances array)))
        harness_syserr();
        /* Unlock harness-example plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
            harness_syserr();
        }
        return -1;
    }

    /* Remove instance from instances array. */
    imgproc_data.count--;
    memmove(imgproc_data.instances + index,
            imgproc_data.instances + index +1,
            (imgproc_data.count    - index) *
            sizeof(unsigned int));

    /* Reallocate instances array. */
    if (0 == imgproc_data.count)
    {
        free(imgproc_data.instances);
        imgproc_data.instances = NULL;
    }
    else if (NULL == (instances = realloc( imgproc_data.instances,
                                           imgproc_data.count *
                                   sizeof( imgproc_data.instances[0]))))
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to reallocate instances array)))
    }
    else
    {
        imgproc_data.instances = instances;
    }
    /* Check for last finalization. */
    if (0 == imgproc_data.count)
    {
        /***************************/
        /* PUT YOUR FINI CODE HERE */
        /***************************/
        imgproc_worklist_destroylist( &worklist);
        imgproc_fini_rmix();
    }

    /* Unlock image processing plug-in mutex. */
    if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
    {
        errno = error;
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
        harness_syserr();
        return -1;
    }
    return 0;
}


/*
```

```
 *  Initializes the necessary rmix parameter and exports the plug-ins.
 *
 *  \return 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_init_rmix"
int imgproc_init_rmix()
{
    int ret;

    IMGPROC_PRINT((
        IMGPROC_INFO(start initialization of RMIX)))

    /* Load RMIX plug-in and initialize RMIX */
    ret = harness_plugins_load( &imgproc_rmix_data.handle_rmix,
                                "libharness-rmix.0", HARNESS_PLUGINS_EXPORT);

    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to load RMIX plug-in)))
        return -1;
    }

    /* Exports Harness while forcing 1000 as specific object handle */
    ret = harness_rmix_export4( &imgproc_rmix_data.localref_harness,
                                "PROTOCOL=RPC OBJECTID=1000");
    if (ret != 0)
    {
        RMIX_LOG((RMIX_WARN(unable to export Harness kernel)))
        return -1;
    }


    /* Exports imgproc plug-in while forcing 1002 as specific object handle */
    ret = rmix_export4( &imgproc_rmix_data.localref_imgproc,
                  "PROTOCOL=RPC OBJECTID=1002", NULL, &rmiximgproc_interface);
    if (ret != 0)
    {
        RMIX_LOG((RMIX_WARN(unable to export imgproc interface)))
        return -1;
    }

    IMGPROC_PRINT((
        IMGPROC_INFO(RMIX initialized and imgproc plug-in exported)))

    return 0;
}


/*
 *  Finalizes rmix and unexports the plug-ins.
 *
 *  \return 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_fini_rmix"
int imgproc_fini_rmix()
{
    int ret;
    int err = 0;

    IMGPROC_PRINT((
```

```
        IMGPROC_INFO(start finalization of RMIX)))

    /* Unexports imgproc plug-in */
    ret = rmix_unexport( &imgproc_rmix_data.localref_imgproc);
    if (ret != 0)
    {
        RMIX_LOG((RMIX_WARN(unable to unexport imgproc interface)))
        err = -1;
    }

    /* Unexports Harness */
    ret = harness_rmix_unexport( &imgproc_rmix_data.localref_harness);
    if (ret != 0)
    {
        RMIX_LOG((RMIX_WARN(unable to unexport Harness kernel)))
        err = -1;
    }

    /* Unloads RMIX plug-in */
    ret = harness_plugins_unload( imgproc_rmix_data.handle_rmix);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to unload RMIX plug-in)))
        err = -1;
    }

    IMGPROC_PRINT((
        IMGPROC_INFO(RMIX finalized and imgproc plug-in unexported)))

    return err;
}


/****************************************************************************
 *
 * next section contains functions for communication
 *
 ****************************************************************************/

/*
 *  Accepts calls for the initialisation of necessary pipeline parameters.
 *  (server-side stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_initpipeline_call"
RMIX_METHOD_CALL(rmiximgproc_initpipeline_call)
{
    int         result = 0;
```

```
    unsigned int   filter;       /* filter which will be used  */
    char          *sourcedir;    /* source images dir          */
    char          *targetdir;    /* target dir for images      */
    char          *successor;    /* successor plug-in          */
    char          *predecessor;  /* predecessor plug-in        */
    char          *ppmnode;      /* node running the PPM       */

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare input. */
    filter      = *(unsigned int*)inary[0];
    sourcedir   =  (char*)        inary[1];
    targetdir   =  (char*)        inary[2];
    successor   =  (char*)        inary[3];
    predecessor =  (char*)        inary[4];
    ppmnode     =  (char*)        inary[5];
```

```
    /* Call method function. */
    result = rmiximgproc_initpipeline( filter, sourcedir, targetdir, successor,
                                       predecessor, ppmnode);

    /* Prepare output. */
    *(int*)(outary[0]) = result;
    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Initializes the necessary pipeline parameters.
 *
 *  \param    filter        Filter which will be used
 *  \param    *sourcedir    Source images dir
 *  \param    *targetdir    Target dir for images
 *  \param    *sucessor     Successor plug-in
 *  \param    *predecessor  Predecessor plug-in
 *  \param    *ppmnode      Node running the PPM
 *  \return                 0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_initpipeline"
int rmiximgproc_initpipeline (unsigned int   filter,
                              char          *sourcedir,
                              char          *targetdir,
                              char          *successor,
                              char          *predecessor,
                              char          *ppmnode)
{

    /* copy the values */
    /* which filter will be used */
    imgproc_data.filter       = filter;

    /* the source directory of the images */
    imgproc_data.sourcedir    = (char*) malloc( sizeof(char) * (strlen(sourcedir)
                                                                + 1) );
    if (imgproc_data.sourcedir == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    strcpy( imgproc_data.sourcedir, sourcedir);

    /* the target directory of the images */
    imgproc_data.targetdir    = (char*) malloc( sizeof(char) * (strlen(targetdir)
                                                                + 1) );
    if (imgproc_data.targetdir == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    strcpy( imgproc_data.targetdir, targetdir);
```

```c
    /* the successor plug-in unit */
    imgproc_data.successor   = (char*) malloc( sizeof(char) * (strlen(successor)
                                                               + 1) );
    if (imgproc_data.successor == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    strcpy( imgproc_data.successor, successor);

    /* the predecessor plug-in unit */
    imgproc_data.predecessor = (char*) malloc( sizeof(char) *
                                         (strlen(predecessor) + 1) );
    if (imgproc_data.predecessor == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    strcpy( imgproc_data.predecessor, predecessor);

    /* the node, which runs the PPM */
    imgproc_data.ppmnode = (char*) malloc( sizeof(char) *
                                         (strlen(ppmnode) + 1) );
    if (imgproc_data.ppmnode == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    strcpy( imgproc_data.ppmnode, ppmnode);

    return 0;
}


/*
 *  Accepts calls for the initialisation of the image counter. (server-side
 *  stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_setimagecounter_call"
RMIX_METHOD_CALL(rmiximgproc_setimagecounter_call)
{
    int          result  = 0;
    unsigned int counter = 0;

#ifdef DEBUG
```

```
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare input. */
    counter = *(unsigned int*)inary[0];

    /* Call method function. */
    result = rmiximgproc_setimagecounter( counter);

    /* Prepare    return 0; output. */
    *(int*)(outary[0]) = result;

    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}
```

```
/*
 *  Forwards an image to the next plug-in.
 *
 *  \param    counter    Number of images to process
 *  \return              0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_setimagecounter"
int rmiximgproc_setimagecounter( unsigned int counter)
{
    rmix_remoteref_t *remoteobj;
    int              ret;

    imgproc_data.imagecounter = counter;

    /* if the plug-in has a successor, the image counter has to be forwarded
       to it */
    if (strcmp(imgproc_data.successor, "nada") != 0)
    {
        /* prepare the remote reference of the successor */
        ret = imgproc_createremoteref( &remoteobj, "1002",
                                       imgproc_data.successor);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))
            return -1;
        }

        /* forward the image counter to the successor */
        ret = rmiximgprocclient_setimagecounter ( remoteobj, counter);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not set image counter of node %s),
                             imgproc_data.successor))
            return -1;

        }

        /* delete the remote reference */
        ret = rmix_remoteref_destroy( &remoteobj);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not destroy remote object references)))
        }
    }
    return 0;
}


/*
 *  Accepts calls for updating the image counter. (server-side stub)
 *
 *  \param   object   The local object.
 *  \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
 *                    dynamically or explicitly set to NULL by this function. In
 *                    the case of variable arrays, the length value is allocated
```

```
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_updateimagecounter_call"
RMIX_METHOD_CALL(rmiximgproc_updateimagecounter_call)
{
    int          result  = 0;
    unsigned int counter = 0;

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */
```

```
    /* Prepare input. */
    counter = *(unsigned int*)inary[0];

    /* Call method function. */
    result = rmiximgproc_updateimagecounter( counter);

    /* Prepare    return 0; output. */
    *(int*)(outary[0]) = result;

    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }

    return 0;
}


/*
 *  Updates the image counter after the pipeline was broken.
 *
 *  \param    counter    Number of images to process
 *  \return              0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_updateimagecounter"
int rmiximgproc_updateimagecounter( unsigned int counter)
{
    imgproc_data.imagecounter = counter;

    return 0;
}


/*
 *  Calls a defined plug-in unit for setting its image counter (client-side
 *  method stub).
 *
 *  \param    *remoteobj  ID of the remote object
 *  \param     counter    Image counter
 *  \return               0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_setimagecounter"
int rmiximgprocclient_setimagecounter ( rmix_remoteref_t *remoteobj,
                                        unsigned int      counter)
{
    const void *inary[1];
    void       *outary[1];
    int         result;

    /* Prepare parameters. */
    inary[0]  = &counter;
    outary[0] = &result;

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_SETIMAGECOUNTER_INDEX, outary, 1,
                         inary, 1))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
```

```
        errno = errno2;
        return -1;
    }

    return result;
}


/*
 *  Accepts calls for the invocation of the pipeline. (server-side stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_invokepipeline_call"
RMIX_METHOD_CALL(rmiximgproc_invokepipeline_call)
{
  int           result;

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/

    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
```

```
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Call method function. */
    result = rmiximgproc_invokepipeline();

    /* Prepare output. */
    *(int*)(outary[0]) = result;

    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Starts the image processing pipeline.
 *
 *  \return 0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_invokepipeline"
int rmiximgproc_invokepipeline ()
{
    int ret;

    /* load the images, use the first filter and forwards the image to a
       possible successor plug-in */
    ret = imgproc_loadimages(imgproc_data.sourcedir);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(execution of image processing pipeline not possible)))
        return -1;
    }

    return 0;
}


/*
 *  Accepts calls, which check if the plug-in was loaded (server-side
 *  method stub).
 *
 *  \param  object  The local object.
 *  \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
 *                   dynamically or explicitly set to NULL by this function. In
 *                   the case of variable arrays, the length value is allocated
 *                   before the call and is set to 0 if its variable array is
 *                   NULL.
 *  \param  outcnt   The output values count.
 *  \param  inary    The input values array. The values array and its containing
 *                   values are not modified or deallocated by this call.
 *  \param  incnt    The input values count.
 *  \return          0 on success or -1 on any error.
 */
RMIX_METHOD_CALL(rmiximgproc_availabilitycheck_call)
{
    int result = 0;

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/

    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }
```

```
#endif /* DEBUG */

    /* Prepare output. */
    *(int*)(outary[0]) = result;

    return 0;
}


/*
 *  Accepts call, which forward images to the plug-in. (server-side stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_passimage_call"
RMIX_METHOD_CALL(rmiximgproc_passimage_call)
{
    int            result;
    char          *name;
    unsigned char *blob;
    unsigned int   bytes;

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/

    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
```

```
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    name  =  (char*)          inary[0];
    bytes = *(unsigned int*) inary[1];
    blob  =  (unsigned char*)inary[2];

    /* Call method function. */
    result = rmiximgproc_passimage( name, bytes, blob);

    /* Prepare output. */
    *(int*)(outary[0]) = result;

    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Processes an image and forwards it to the next plug-in unit if possible.
 *
 *  \param  *name   Name of the image file
 *  \param   size   Size of image data
 *  \param  *blob   Image data
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_passimage"
int rmiximgproc_passimage (char          *name,
                           size_t         size,
                           unsigned char *blob)
{
    char              *storename;
    char              *tmpfile;
    rmix_remoteref_t *remoteobj;
    rmix_remoteref_t *remoteobj_ppm;

    int error;
    int ret;

    /* process filter */
```

```
ret = filters[imgproc_data.filter](&blob, &size);
if (ret != 0)
{
    IMGPROC_PRINT((
        IMGPROC_WARN(could not process image filter)))
    return -1;
}

/* if a successor plug-in exists, forward the image to it */
if (strcmp(imgproc_data.successor, "nada") != 0)
{
    /* prepare the remote reference of the successor */
    ret = imgproc_createremoteref( &remoteobj, "1002",
                                   imgproc_data.successor);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not create remote object references)))

        /* prepare the remote reference of the PPM node */
        ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                       imgproc_data.ppmnode);
        if(ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))
            return -1;
        }

        /* call the PPM for repairing the pipeline */
        ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                        imgproc_data.successor);
        if(ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not repair pipe)))
            rmix_remoteref_destroy( &remoteobj_ppm);
            return -1;
        }

        rmix_remoteref_destroy( &remoteobj_ppm);
        ret = imgproc_createremoteref( &remoteobj, "1002",
                                       imgproc_data.successor);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object
                        references)))
            return -1;
        }

    }

    /* pass the image with an onway RPC call */
    ret = rmiximgprocclient_passimage_oneway ( remoteobj, name, blob, size);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not call remote object for passing image)))

        /* prepare the remote reference of the PPM node */
        ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                       imgproc_data.ppmnode);
```

```
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object references)))
                return -1;
            }

            /* call the PPM for repairing the pipeline */
            ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                            imgproc_data.successor);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not repair pipe)))
                rmix_remoteref_destroy( &remoteobj_ppm);
                return -1;
            }

            rmix_remoteref_destroy( &remoteobj_ppm);
            ret = imgproc_createremoteref( &remoteobj, "1002",
                                            imgproc_data.successor);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object
                                references)))
                return -1;
            }
            ret = rmiximgprocclient_passimage_oneway ( remoteobj, name, blob,
                                                        size);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not call remote object for passing
                            image)))
                return -1;
            }
        }

        /* destroy the remote reference */
        ret = rmix_remoteref_destroy( &remoteobj);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not destroy remote object references)))
        }


        /* if image was sent successfully, store it in the internal list until
           an acknowledgment arrives */

        /* the access to the list is controlled by a mutex */
        /* Lock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to lock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        /* store the image if it was not stored yet */
        ret = imgproc_worklist_find_elementposition( name, worklist);
```

```
        if (ret == −1)
            imgproc_worklist_insert( name, size, blob, &worklist);

        /* Unlock image processing plug−in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug−in mutex)))
            harness_syserr();
            return −1;
        }

        IMGPROC_PRINT((
            IMGPROC_INFO(image %s successfully sent to %s) ,name
                        ,imgproc_data.successor))
    }
    /* if there is no successor plug−in, store the processed image */
    else
    {
        /* construct the whole path of the image file */
        storename = (char*)malloc( sizeof(char) *
                                (strlen(imgproc_data.targetdir) +2));
        if (storename == NULL)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(allocating of memory failed)))
            return −1;
        }
        strcpy(storename, imgproc_data.targetdir);
        strcat(storename, "/");

        tmpfile = storename;

        /* put image file name and path together */
        storename = (char*)malloc( sizeof(char) * (strlen(tmpfile) +
                                 strlen(name) +1 ));
        if (storename == NULL)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(allocating of memory failed)))
            return −1;
        }
        strcpy( storename, tmpfile);
        strcat( storename, name);

        /* store the image */
        imgproc_write_image( blob, size, storename);

        IMGPROC_PRINT((
            IMGPROC_INFO(image %s successfully stored),name))

        /* this section was used to simulate a broken pipeline and calling the
           PPM for restoration */
        /*
        if (testswitch == 0)
        {
            imgproc_createremoteref( &remoteobj, "1003",imgproc_data.ppmnode);
            rmixppmclient_repairpipe ( remoteobj,imgproc_data.successor);
            rmix_remoteref_destroy( &remoteobj);
            testswitch = 1;
        }
        */
```

```
        /* prepare the remote reference for the predecessor */
        ret = imgproc_createremoteref( &remoteobj, "1002",
                                       imgproc_data.predecessor);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))

            /* prepare the remote reference for the PPM node */
            ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                           imgproc_data.ppmnode);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object references)))
                return -1;
            }

            /* call the PPM for pipeline restoration */
            ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                            imgproc_data.predecessor);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not repair pipe)))
                rmix_remoteref_destroy( &remoteobj_ppm);
                return -1;
            }

            rmix_remoteref_destroy( &remoteobj_ppm);
            ret = imgproc_createremoteref( &remoteobj, "1002",
                                           imgproc_data.predecessor);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object
                                 references)))
                return -1;
            }
        }

        /* inform the predecessor about a successfully stored image */
        ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
        if (ret != 0)
        {

            IMGPROC_PRINT((
                IMGPROC_WARN(could not call remote object for passing image)))

            /* prepare the remote reference of the PPM node */
            ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                           imgproc_data.ppmnode);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object references)))
                return -1;
            }

            /* call the PPM for pipeline restoration */
            ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                            imgproc_data.predecessor);
```

```
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not repair pipe)))
                rmix_remoteref_destroy( &remoteobj_ppm);
                return -1;
            }

            rmix_remoteref_destroy( &remoteobj_ppm);
            ret = imgproc_createremoteref( &remoteobj, "1002",
                                            imgproc_data.predecessor);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object
                                    references)))
                return -1;
            }

            /* resend the message */
            ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not call remote object for passing
                                image)))
                return -1;
            }
        }

        /* destroy the remote reference */
        ret = rmix_remoteref_destroy( &remoteobj);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not destroy remote object references)))
        }

        FREE(tmpfile);
        FREE(storename);

        /* Lock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to lock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        /* store the image if it was not stored yet */
        ret = imgproc_worklist_find_elementposition( name, worklist);
        if (ret == -1)
        {
            imgproc_worklist_insert( name, 0, NULL, &worklist);

            /* one more image was processed, decrease the counter */
            imgproc_decrease_imagecounter();
        }

        /* Unlock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
```

```
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug−in mutex)))
            harness_syserr();
            return −1;
        }

        IMGPROC_PRINT((
            IMGPROC_INFO(acknowledgment for %s successfully sent to %s) ,name
                          ,imgproc_data.predecessor))

    }

    /* check if the iamge counter is zero and all images were processed */
    if (imgproc_data.imagecounter == 0)
    {
        /* if yes shutdown the plug−in and the Harness kernel */
        harness_kernel_shutdown();
    }
    return(0);
}


/*
 *  Passes an image to the next plug−in (client−side method stub).
 *
 *  \param  *remoteobj  ID of the remote object
 *  \param  *name       Name of the image file
 *  \param  *blob       Image data
 *  \param   size       Size of the image data
 *  \return             0 on success or −1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_passimage_oneway"
int rmiximgprocclient_passimage_oneway ( rmix_remoteref_t *remoteobj,
                                         char              *name,
                                         unsigned char     *blob,
                                         unsigned int       size)
{
    const void *inary[3];

    /* pack data */
    inary[0] =  name;
    inary[1] = &size;
    inary[2] =  blob;

    /* Invoke remote object method by using a one way invocation. */
    if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_PASSIMAGE_INDEX, inary, 3))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return −1;
    }

    return 0;
}


/*
 *  Accepts acknowledgment calls that an image was successfully stored.
 *  (server−side stub)
```

```
 *
 * \param   object   The local object.
 * \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
 *                   dynamically or explicitly set to NULL by this function. In
 *                   the case of variable arrays, the length value is allocated
 *                   before the call and is set to 0 if its variable array is
 *                   NULL.
 * \param   outcnt   The output values count.
 * \param   inary    The input values array. The values array and its containing
 *                   values are not modified or deallocated by this call.
 * \param   incnt    The input values count.
 * \return           0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_imageprocessed_call"
RMIX_METHOD_CALL(rmiximgproc_imageprocessed_call)
{
    int    result = 0;

    char *imgname;    /* image name */

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
```

```
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare input. */
    imgname = (char*)inary[0];

    /* Call method function. */
    result = rmiximgproc_imageprocessed( imgname);

    /* Prepare output. */
    *(int*)(outary[0]) = result;
    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Deletes a processed image from the internal worklist. If all images are
 *  processed the plug-in is shutdown. If there is a predecessor plug-in unit,
 *  the message is forwarded to it.
 *
 *  \param    *name  Name of the image.
 *  \return          0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_imageprocessed"
int rmiximgproc_imageprocessed (char* name)
{
    int error;
    int ret;
    rmix_remoteref_t *remoteobj;
    rmix_remoteref_t *remoteobj_ppm;

    /* if a predecessor plug-in exists, forward the name to it */
    if (strcmp(imgproc_data.predecessor, "nada") != 0)
    {
        /* prepare the remote reference of the predecessor */
        ret = imgproc_createremoteref( &remoteobj, "1002",
                                       imgproc_data.predecessor);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))

            /* prepare the remote reference of the PPM node */
            ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                           imgproc_data.ppmnode);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object references)))
                return -1;
            }
```

```
            /* call the PPM for pipeline restoration */
            ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                            imgproc_data.predecessor);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not repair pipe)))
                rmix_remoteref_destroy( &remoteobj_ppm);
                return -1;
            }

            rmix_remoteref_destroy( &remoteobj_ppm);
            ret = imgproc_createremoteref( &remoteobj, "1002",
                                            imgproc_data.predecessor);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object
                                references)))
                return -1;
            }
        }

        /* pass the acknowledgment with an onway RPC call */
        ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not call remote object for passing image)))

            /* prepare the remote reference of the PPM call */
            ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                            imgproc_data.ppmnode);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object references)))
                return -1;
            }

            /* call the PPM for pipeline restoration */
            ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                            imgproc_data.predecessor);
            if(ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not repair pipe)))
                rmix_remoteref_destroy( &remoteobj_ppm);
                return -1;
            }

            rmix_remoteref_destroy( &remoteobj_ppm);
            ret = imgproc_createremoteref( &remoteobj, "1002",
                                            imgproc_data.predecessor);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object
                                references)))
                return -1;
            }

            /* resend the message */
```

```
            ret = rmiximgprocclient_imageprocessed_oneway ( remoteobj, name);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not call remote object for passing
                                 image)))
                return -1;
            }
        }

        /* destroy the remote reference */
        ret = rmix_remoteref_destroy( &remoteobj);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not destroy remote object references)))

        }

        /* delete the image from the internal backup list */
        /* Lock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to lock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        ret = imgproc_worklist_find_elementposition( name, worklist);
        if (ret != -1)
        {
            imgproc_worklist_delete( name, &worklist);
            /* one more image was processed, decrease the counter */
            imgproc_decrease_imagecounter();
        }

        /* Unlock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        IMGPROC_PRINT((
            IMGPROC_INFO(acknowledgement for %s successfully sent to %s) ,name
                         ,imgproc_data.predecessor))
    }
    else
    {
        /* if it is the first pipeline unit, only delete it from the backup
           list */

        /* Lock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to lock image processing plug-in mutex)))
            harness_syserr();
            return -1;
```

```
        }

        ret = imgproc_worklist_find_elementposition( name, worklist);
        if (ret != -1)
        {
            imgproc_worklist_delete( name, &worklist);
            /* one more image was processed, decrease the counter */
            imgproc_decrease_imagecounter();
        }

        /* Unlock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }
    }

    /* check if the counter is zero and all images were processed */
    if (imgproc_data.imagecounter == 0)
    {
        /* if yes shutdown the plug-in and the Harness kernel */
        harness_kernel_shutdown();
    }

    return 0;
}


/*
 * Sends an acknowledgment for a successfully stored iamge (client-side
 * method stub).
 *
 * \param  *remoteobj  ID of the remote object
 * \param  *name       Name of the image file
 * \return             0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgprocclient_imageprocessed_oneway"
int rmiximgprocclient_imageprocessed_oneway ( rmix_remoteref_t *remoteobj,
                                              char             *name)
{
    const void *inary[1];

    /* pack data */
    inary[0] =  name;

    /* Invoke remote object method by using a one way invocation. */
    if (0 != rmix_oneway(remoteobj, &rmiximgprocclient_interface,
                         RMIXIMGPROC_METHODS_IMAGEPROCESSED_INDEX, inary, 1))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return 0;
}
```

```
/*
 *  Accepts calls for updating the predecessor entry of the plug-in.
 *  (server-side stub)
 *
 *  \param  object  The local object.
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef   __FUNC__
#define __FUNC__ "rmiximgproc_updatesuccessor_call"
RMIX_METHOD_CALL(rmiximgproc_updatesuccessor_call)
{
    int    result = 0;

    char *predecessor;    /* successor name          */

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
```

```
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare input. */
    predecessor = (char*)inary[0];

    /* Call method function. */
    result = rmiximgproc_updatesuccessor( predecessor);

    /* Prepare output. */
    *(int*)         (outary[0]) = result;
    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Updates the predecessor entry of the plug-in.
 *
 *  \param   *predecessor  Name of the predecessor plug-in.
 *  \return                0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_updatesuccessor"
int rmiximgproc_updatesuccessor (char *predecessor)
{
    /* delete the old entry */
    FREE(imgproc_data.predecessor);

    /* allocate memory for the new entry */
    imgproc_data.predecessor = (char*)malloc( sizeof(char) *
                                             (strlen(predecessor) + 1));
    if (imgproc_data.predecessor == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    /* store the new entry */
    strcpy( imgproc_data.predecessor, predecessor);

    return 0;
}


/*
 *  Accepts call for updating the successor entry of the plug-in and returns the
 *  current image counter. (server-side stub)
 *
 *  \param  object  The local object.
```

```
 *  \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
 *                  dynamically or explicitly set to NULL by this function. In
 *                  the case of variable arrays, the length value is allocated
 *                  before the call and is set to 0 if its variable array is
 *                  NULL.
 *  \param  outcnt  The output values count.
 *  \param  inary   The input values array. The values array and its containing
 *                  values are not modified or deallocated by this call.
 *  \param  incnt   The input values count.
 *  \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_updatepredecessor_call"
RMIX_METHOD_CALL(rmiximgproc_updatepredecessor_call)
{
    int   result = 0;

    char          *successor;   /* successor name        */
    unsigned int  counter;      /* current image counter */

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
```

```
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return -1;
    }
#endif /* DEBUG */

    /* Prepare input. */
    successor = (char*)inary[0];

    /* Call method function. */
    result = rmiximgproc_updatepredecessor( successor, &counter);

    /* Prepare output. */
    *(int*)          (outary[0]) = result;
    *(unsigned int*)(outary[1]) = counter;
    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }

    return 0;
}


/*
 *  Updates the successor entry of the plug-in and returns the current image
 *  counter.
 *
 *  \param   *successor   Name of the successor plug-in.
 *  \param   *counter     Current image counter.
 *  \return               0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_updatepredecessor"
int rmiximgproc_updatepredecessor( char          *successor,
                                   unsigned int *counter)
{
    /* delete the old entry */
    FREE(imgproc_data.successor);

    /* allocate memory for the new entry */
    imgproc_data.successor = (char*)malloc( sizeof(char) * (strlen(successor)
                                                        + 1));
    if (imgproc_data.successor == NULL)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not allocate memory)))
        return -1;
    }
    /* store the new entry */
    strcpy( imgproc_data.successor, successor);

    /* return the image counter */
    *counter = imgproc_data.imagecounter;

    return 0;
}


/*
 *  Accepts calls for sending the images in the backup list again to the
```

```
 * successor (server-side stub).
 *
 * \param  object  The local object.
 * \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
 *                 dynamically or explicitly set to NULL by this function. In
 *                 the case of variable arrays, the length value is allocated
 *                 before the call and is set to 0 if its variable array is
 *                 NULL.
 * \param  outcnt  The output values count.
 * \param  inary   The input values array. The values array and its containing
 *                 values are not modified or deallocated by this call.
 * \param  incnt   The input values count.
 * \return         0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_sendworklist_call"
RMIX_METHOD_CALL(rmiximgproc_sendworklist_call)
{
    int   result = 0;

#ifdef DEBUG
    /* Check object parameter. */
/*
    if (NULL == object)
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(object parameter is null)))
        errno = EINVAL;
        return -1;
    }
*/
    /* Check outary parameter. */
    if ((NULL == outary)&&(0 != outcnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check outcnt parameter. */
    if ((0 == outcnt)&&(NULL != outary))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(outcnt parameter is zero)))
        errno = EINVAL;
        return -1;
    }

    /* Check inary parameter. */
    if ((NULL == inary)&&(0 != incnt))
    {
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(inary parameter is null)))
        errno = EINVAL;
        return -1;
    }

    /* Check incnt parameter. */
    if ((0 == incnt)&&(NULL != inary))
    {
```

```
        errno = EINVAL;
        RMIX_LOG((RMIX_WARN(incnt paramet is zero)))
        errno = EINVAL;
        return −1;
    }
#endif /* DEBUG */

    /* Call method function. */
    result = rmiximgproc_sendworklist();

    /* Prepare output. */
    *(int*)(outary[0]) = result;
    /* Reset errno if needed. */
    if (0 != result)
    {
        errno = 0;
    }
    return 0;
}


/*
 *  Sends again the images in the worklist to the successor plug−in.
 *
 *  \return 0 on success or −1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "rmiximgproc_sendworklist"
int rmiximgproc_sendworklist()
{
    int error;
    int ret;
    worklist_t *tmpworklist;
    worklist_t *worklistcopy = NULL;
    rmix_remoteref_t *remoteobj;

    /* avoid that other threads change the backup list */

    /* Lock image processing plug−in mutex. */
    if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex)))
    {
        errno = error;
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to lock image processing plug−in mutex)))
        harness_syserr();
        return −1;
    }

    /* copy liste */

    tmpworklist = worklist;
    while(tmpworklist != NULL)
    {

        imgproc_worklist_insert( tmpworklist−>name, tmpworklist−>size,
                                 tmpworklist−>blob, &worklistcopy);


        tmpworklist = tmpworklist−>next;
    }

    /* Unlock image processing plug−in mutex. */
    if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
```

```
    {
        errno = error;
        IMGPROC_PRINT((
            IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
        harness_syserr();
        return -1;
    }

    /* debug print out */
    imgproc_worklist_printlist(worklistcopy);

    /* resend the images but do not put them again in the list */
    tmpworklist = worklistcopy;

    /* prepare the remote reference of the successor */
    ret = imgproc_createremoteref( &remoteobj, "1002", imgproc_data.successor);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not create remote object)))
        return -1;
    }

    while(tmpworklist != NULL)
    {
        /* forward the image */
        IMGPROC_PRINT((
            IMGPROC_INFO(resending image %s),tmpworklist->name))
        ret = rmiximgprocclient_passimage_oneway ( remoteobj, tmpworklist->name,
                                                   tmpworklist->blob,
                                                   tmpworklist->size);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not send image)))
            tmpworklist = tmpworklist->next;
            continue;
        }
        tmpworklist = tmpworklist->next;
    }

    /* destroy the remote reference */
    ret = rmix_remoteref_destroy( &remoteobj);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not destroy remote object references)))
    }

    /* delete the copy of the backup list */
    imgproc_worklist_destroylist( &worklistcopy);

    return 0;
}


/*
 *  Call the Parallel Plug-in Manager for repairing the pipeline (client-side
 *  method stub).
 *
 *  \param    *remoteobj  ID of the remote object
 *  \param    *node       Name of the not reachable node
 *  \return               0 on success or -1 on any error
```

```
 */
#undef  __FUNC__
#define __FUNC__ "rmixppmclient_repairpipe"
int rmixppmclient_repairpipe ( rmix_remoteref_t *remoteobj,
                               char             *node)
{
    const void *inary[1];
    void       *outary[1];
    int         result;

    /* Prepare parameters. */
    inary[0]  =  node;
    outary[0] = &result;

    /* Invoke remote object method. */
    if (0 != rmix_invoke(remoteobj, &rmixppmclient_interface,
                         RMIXPPM_METHODS_REPAIRPIPE_INDEX, outary, 1,
                         inary, 1))
    {
        int errno2 = errno;
        RMIX_LOG((RMIX_WARN(unable to invoke remote object method)))
        errno = errno2;
        return -1;
    }

    return result;
}


/*
 *  Creates a remote reference specified by objectid and node. The RPC protocol
 *  is used.
 *
 *  \param    **remoteobj  ID of the remote object
 *  \param     *objectid   ID of the exported object
 *  \param     *node       Name of the node
 *  \return                0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_createremoteref"
int imgproc_createremoteref( rmix_remoteref_t **remoteobj,
                             char              *objectid,
                             char              *node)
{
    int   ret;
    char *remoteparameters;

    /* prepare parameter for creating remote reference */
    remoteparameters = (char*)malloc( sizeof(char) *
                         (strlen("PROTOCOL=RPC OBJECTID=%s HOST=%s") +
                          strlen(objectid) +
                          strlen(node) - 3 ));
    if (remoteparameters == NULL)
    {
        IMGPROC_PRINT((
                IMGPROC_WARN(could not allocate memory)))
            return -1;
    }
    sprintf( remoteparameters, "PROTOCOL=RPC OBJECTID=%s HOST=%s",
            objectid, node);

    /* create the remote reference */
    ret = rmix_remoteref_create6( &(*remoteobj), remoteparameters);
```

```
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not create remote object references)))
        return -1;
    }

    FREE(remoteparameters);
    return 0;
}


/*****************************************************************************
 *
 * next section contains functions for processing the images
 *
 *****************************************************************************/

/*
 *  Decreases the image counter by 1.
 *
 *  \return 0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_decrease_imagecounter"
int imgproc_decrease_imagecounter()
{
    /* decrease image counter */
    imgproc_data.imagecounter--;

    return 0;
}


/*
 *  Rotates an image by 90 degrees.
 *
 *  \param  **blob   Char array containing the image data.
 *  \param   *size   Size of the char array.
 *  \return          0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_filter_rotation"
int imgproc_filter_rotation( unsigned char **blob,
                             size_t         *size)
{
    MagickBooleanType status;
    MagickWand *magick_wand;
    PixelWand *background_color;

    magick_wand = NewMagickWand();
    background_color = NewPixelWand();

    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, *blob, *size);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* get the background color, which is needed to fill empty triangles left
       over from rotating the image */
```

```
    status = MagickGetImageBackgroundColor( magick_wand, background_color);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* rotate image by 90 degree */
    status = MagickRotateImage( magick_wand, background_color, 90);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* recreate blob */
    *blob = MagickGetImageBlob( magick_wand, size);

    DestroyMagickWand(magick_wand);

    return 0;
}


/*
 *  Puts an oilpainting filter on an image.
 *
 *  \param  **blob   Char array containing the image data.
 *  \param   *size   Size of the char array.
 *  \return          0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_filter_oilpainting"
int imgproc_filter_oilpainting( unsigned char **blob,
                                size_t         *size)
{
    MagickBooleanType status;
    MagickWand *magick_wand;

    magick_wand = NewMagickWand();

    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, *blob, *size);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    status = MagickOilPaintImage(magick_wand, 5);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* recreate blob */
    *blob = MagickGetImageBlob( magick_wand, size);

    DestroyMagickWand(magick_wand);

    return 0;
}
```

```
/*
 *  Swirls the center of an image.
 *
 *  \param  **blob   Char array containing the image data.
 *  \param   *size   Size of the char array.
 *  \return          0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_filter_swirl"
int imgproc_filter_swirl( unsigned char **blob,
                          size_t         *size)
{
    MagickBooleanType status;
    MagickWand *magick_wand;

    magick_wand = NewMagickWand();

    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, *blob, *size);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    status = MagickSwirlImage(magick_wand, 180);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* recreate blob */
    *blob = MagickGetImageBlob( magick_wand, size);

    DestroyMagickWand(magick_wand);

    return 0;
}


/*
 *  Negates an image.
 *
 *  \param  **blob   Char array containing the image data.
 *  \param   *size   Size of the char array.
 *  \return          0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_filter_negate"
int imgproc_filter_negate( unsigned char **blob,
                           size_t         *size)
{
    MagickBooleanType status;
    MagickWand *magick_wand;

    magick_wand = NewMagickWand();

    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, *blob, *size);
    if (status == MagickFalse)
```

```
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    status = MagickNegateImage(magick_wand, MagickFalse);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* recreate blob */
    *blob = MagickGetImageBlob( magick_wand, size);

    DestroyMagickWand(magick_wand);

    return 0;
}


/*
 *  Puts a solarize filter on an image.
 *
 *  \param  **blob   Char array containing the image data.
 *  \param   *size   Size of the char array.
 *  \return          0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_filter_solarize"
int imgproc_filter_solarize( unsigned char **blob,
                             size_t         *size)
{
    MagickBooleanType status;
    MagickWand *magick_wand;

    magick_wand = NewMagickWand();

    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, *blob, *size);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    status = MagickSolarizeImage( magick_wand, 80);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    /* recreate blob */
    *blob = MagickGetImageBlob( magick_wand, size);

    DestroyMagickWand(magick_wand);

    return 0;
}


/*
```

```
 *  Stores an image.
 *
 *  \param  **blob      Char array containing the image data.
 *  \param   *size      Size of the char array.
 *  \param   *filename  Name of the new image file.
 *  \return             0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_write_image"
int imgproc_write_image( unsigned char *blob,
                         size_t         size,
                         char          *filename)
{
    MagickBooleanType status;
    MagickWand *magick_wand;

    magick_wand = NewMagickWand();

    /* create image from blob */
    status = MagickReadImageBlob( magick_wand, blob, size);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    status = MagickWriteImages( magick_wand, filename, MagickTrue);
    if (status == MagickFalse)
    {
        ThrowWandException(magick_wand);
        return -1;
    }

    DestroyMagickWand(magick_wand);

    return 0;
}


/*
 *  Returns the number of files found in a directory.
 *
 *  \param  *char   Directory name.
 *  \return         Number of files
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_get_files"
int imgproc_get_files( char *dir)
{
    int images = 0;

    DIR* DirectoryPointer;
    struct dirent* dp;

    /* open the directory stream */
    DirectoryPointer = opendir(dir);

    /* count the numbers of files */
    for (dp = readdir(DirectoryPointer); dp!=0; dp=readdir(DirectoryPointer) )
    {
        if (strcmp(dp->d_name,".")!=0 && strcmp(dp->d_name,"..")!=0)
        {
            images++;
```

```
        }
    }

    /* close the directory stream */
    closedir(DirectoryPointer);

    IMGPROC_PRINT((
        IMGPROC_INFO(%d images found), images))
    return images;
}


/*
 *  Tries to load the image files in the specified directory and additional
 *  files.
 *
 *  \param   *imgdir     Directory containing the image files.
 *  \return              0 on success or -1 on any error.
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_loadimages"
int imgproc_loadimages( char  *imgdir)
{
    int                 images;
    int                 ret;
    char               *imgfile;            /* path + filename */
    char               *imgname;            /* image name */
    char               *tmpfile;

    rmix_remoteref_t *remoteobj;

    DIR                *DirectoryPointer;  /* directorystream object */
    struct dirent      *dp;

    imgfile = NULL;
    tmpfile = NULL;

    /* get the number of image files */
    images = imgproc_get_files(imgdir);

    if (images == 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(no images in source directory)))
        return -1;
    }
    else
    {
        /* set the image counter of all plug-in units by forwarding it*/
        if (strcmp(imgproc_data.successor, "nada") != 0)
        {
            /* set the parameters for creating the remote reference to
               the successor */
            ret = imgproc_createremoteref( &remoteobj, "1002",
                                            imgproc_data.successor);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not create remote object)))
                return -1;
            }

            /* forward the image counter */
```

```
            ret = rmiximgprocclient_setimagecounter ( remoteobj, images);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not set image counter)))
                return -1;
            }

            /* destroy the remote reference */
            ret = rmix_remoteref_destroy( &remoteobj);
            if (ret != 0)
            {
                IMGPROC_PRINT((
                    IMGPROC_WARN(could not destroy remote object references)))
            }

        }
        /* set the own image counter */
        imgproc_data.imagecounter = images;

        /* start with processing the images */

        /* open a directorystream */
        DirectoryPointer = opendir( imgdir);
        for (dp=readdir(DirectoryPointer); dp!=0; dp=readdir(DirectoryPointer))
        {
            if (strcmp(dp->d_name,".")!=0 && strcmp(dp->d_name,"..")!=0)
            {
                IMGPROC_PRINT((
                    IMGPROC_INFO(reading file %s), dp->d_name))

                /* copy image name */
                imgname = (char*)malloc( sizeof(char) * (strlen(dp->d_name) +1));
                if (imgname == NULL)
                {
                    IMGPROC_PRINT((
                        IMGPROC_WARN(allocating of memory failed)))
                    return -1;
                }
                strcpy(imgname, dp->d_name);

                /* construct the whole path of the image */
                imgfile = (char*)malloc( sizeof(char) * (strlen(imgdir) +2));
                if (imgfile == NULL)
                {
                    IMGPROC_PRINT((
                        IMGPROC_WARN(allocating of memory failed)))
                    return -1;
                }
                strcpy(imgfile, imgdir);
                strcat(imgfile, "/");

                tmpfile = imgfile;

                /* add the filename to the path */
                imgfile = (char*)malloc( sizeof(char) * (strlen(tmpfile) +
                                          strlen(dp->d_name) +1 ));
                if (imgfile == NULL)
                {
                    IMGPROC_PRINT((
                        IMGPROC_WARN(allocating of memory failed)))
                    return -1;
                }
```

```
                    strcpy( imgfile , tmpfile );
                    strcat( imgfile , dp->d_name );

                    FREE( tmpfile );

                    IMGPROC_PRINT((
                        IMGPROC_INFO( full filename is %s), imgfile))

                    /* process the image and forward it */
                    ret = imgproc_create_magickwand(imgfile , imgname);
                    if (ret != 0)
                    {
                        IMGPROC_PRINT((
                            IMGPROC_WARN(cannot start processing the first filter)))
                        FREE(imgname);
                        FREE(imgfile);
                        return -1;
                    }

                    FREE(imgname);
                    FREE(imgfile);
                }
            }
            closedir(DirectoryPointer);
        }

        /* check if the counter is zero and all images were processed */
        if (imgproc_data.imagecounter == 0)
        {
            /* if yes shutdown the plug-in and the Harness kernel */
            harness_kernel_shutdown();
        }

        return 0;
}


/*
 *  Processes an image and forwards it to the next plug-in unit. This function
 *  is only executed by the first unit in the pipeline.
 *
 *  \param  *imgfile   File name of the image + path
 *  \param  *imgname   File name
 *  \return            0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_create_magickwand"
int imgproc_create_magickwand( char *imgfile , char *imgname)
{
    MagickBooleanType  status;
    MagickWand         *magick_wand;

    unsigned char      *blob;
    size_t              size;
    char               *storename;
    char               *tmpfile;

    rmix_remoteref_t   *remoteobj;
    rmix_remoteref_t   *remoteobj_ppm;

    int                 ret;
    int error;

```

```
magick_wand = NewMagickWand();

/* Read an image. */
status = MagickReadImage( magick_wand, imgfile);
if (status == MagickFalse)
{
    ThrowWandException(magick_wand);
    return -1;
}

/* create image blob */
blob = MagickGetImageBlob( magick_wand, &size);

DestroyMagickWand(magick_wand);

/* use first filter */
ret = filters[imgproc_data.filter](&blob, &size);
if (ret != 0)
{
    IMGPROC_PRINT((
        IMGPROC_WARN(error while processing image filter)))
    return -1;
}


/* if there is a successor plug-in, forward the image to it */
if (strcmp(imgproc_data.successor, "nada") != 0)
{
    /* create remote reference for the successor */
    ret = imgproc_createremoteref( &remoteobj, "1002",
                                  imgproc_data.successor);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not create remote object references)))

        /* create remote refernce of the PPM node */
        ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                      imgproc_data.ppmnode);
        if(ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))
            return -1;
        }

        /* call the PPM for pipeline restoration */
        ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                      imgproc_data.successor);
        if(ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not repair pipe)))
            rmix_remoteref_destroy( &remoteobj_ppm);
            return -1;
        }

        rmix_remoteref_destroy( &remoteobj_ppm);
        ret = imgproc_createremoteref( &remoteobj, "1002",
                                      imgproc_data.successor);
        if (ret != 0)
        {
            IMGPROC_PRINT((
```

```
                IMGPROC_WARN(could not create remote object
                                references)))
            return -1;
        }
    }

    /* forward the processed image to the next plug-in unit */
    ret = rmiximgprocclient_passimage_oneway( remoteobj, imgname,blob,size);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not call remote object for passing image)))

        /* create remote reference of the PPM node */
        ret = imgproc_createremoteref( &remoteobj_ppm, "1003",
                                        imgproc_data.ppmnode);
        if(ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object references)))
            return -1;
        }

        /* call the PPM for pipeline restoration */
        ret = rmixppmclient_repairpipe( remoteobj_ppm,
                                        imgproc_data.successor);
        if(ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not repair pipe)))
            rmix_remoteref_destroy( &remoteobj_ppm);
            return -1;
        }

        rmix_remoteref_destroy( &remoteobj_ppm);
        ret = imgproc_createremoteref( &remoteobj, "1002",
                                        imgproc_data.successor);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not create remote object
                                references)))
            return -1;
        }

        /* resend the image */
        ret = rmiximgprocclient_passimage_oneway ( remoteobj, imgname, blob,
                                                    size);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not call remote object for passing
                            image)))
            return -1;
        }

    }

    ret = rmix_remoteref_destroy( &remoteobj);
    if (ret != 0)
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(could not destroy remote object references)))
```

```
        }

        /* Lock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_lock(&imgproc_data.mutex))) {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to lock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        /* store the image in the backup list if it is not stored yet */
        ret = imgproc_worklist_find_elementposition( imgname, worklist);
        if (ret == -1)
            imgproc_worklist_insert( imgname, size, blob, &worklist);

        /* Unlock image processing plug-in mutex. */
        if (0 != (error = pthread_mutex_unlock(&imgproc_data.mutex)))
        {
            errno = error;
            IMGPROC_PRINT((
                IMGPROC_WARN(unable to unlock image processing plug-in mutex)))
            harness_syserr();
            return -1;
        }

        IMGPROC_PRINT((
            IMGPROC_INFO(image %s successfully sent to %s),imgname
                        ,imgproc_data.successor))


        /* testing the reload of a plug-in */
        /*
        if (testswitch == 0)
        {
            imgproc_createremoteref( &remoteobj, "1003",imgproc_data.ppmnode);
            rmixppmclient_repairpipe ( remoteobj,imgproc_data.successor);
            rmix_remoteref_destroy( &remoteobj);
            testswitch = 1;
        }
        */

    }
    /* it there is no successor, store the processed image */
    else
    {
        /* construct the path */
        storename = (char*)malloc( sizeof(char) *
                                   (strlen(imgproc_data.targetdir) +2));
        if (storename == NULL)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(allocating of memory failed)))
            return -1;
        }
        strcpy(storename, imgproc_data.targetdir);
        strcat(storename, "/");

        tmpfile = storename;

        /* add the filename to the path */
        storename = (char*)malloc( sizeof(char) * (strlen(tmpfile) +
```

```
                                          strlen(imgname) +1 ));
        if (storename == NULL)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(allocating of memory failed)))
            return -1;
        }
        strcpy( storename, tmpfile);
        strcat( storename, imgname);


        ret = imgproc_write_image( blob, size, storename);
        if (ret != 0)
        {
            IMGPROC_PRINT((
                IMGPROC_WARN(could not store image file %s), storename))
            FREE(tmpfile);
            FREE(storename);
            return -1;
        }

        FREE(tmpfile);
        FREE(storename);

        /* if the pipeline only consists of the first plug-in, first it is no
           pipeline and second, the images do not have to be stored */

        /* one more image was processed, decrease the counter */
        imgproc_decrease_imagecounter();
    }
    return(0);
}


/****************************************************************************
 *
 * next section contains functions for accessing the image  backup list
 *
 ****************************************************************************/

/*
 *  Inserts an element in the backup list.
 *
 *  \param   *name             Name of the image
 *  \param    size             Size of the image
 *  \param   *blob             Data of the image
 *  \param  **list_pointer     Pointer to the linked list
 *  \return                    0 on success or -1 on any error
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_worklist_insert"
int imgproc_worklist_insert( char            *name,
                             size_t           size,
                             unsigned char   *blob,
                             worklist_t     **list_pointer)
{
    int i;
    worklist_t *new_element;
    worklist_t *tmp_element;

    /* allocate memory for the new element */
    new_element = (worklist_t*)malloc(sizeof(worklist_t));
    if ( new_element == NULL )
```

```
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(allocating of memory for new list element not
                possible)))
        return -1;
    }

    /* allocate memory for the name string of the new element */
    new_element->name = (char*)malloc(sizeof(char)*(strlen(name)+1));
    if ( new_element->name == NULL )
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(allocating of memory for the name string of the new
                list element not possible)))
        FREE(new_element);
        return -1;
    }

    /* allocate memory for the image data of the new element */
    new_element->blob = (unsigned char*) malloc(sizeof(unsigned char) * size);
    if ( new_element->blob == NULL )
    {
        IMGPROC_PRINT((
            IMGPROC_WARN(allocating of memory for the name string of the new
                list element not possible)))
        FREE(new_element->name);
        FREE(new_element);
        return -1;
    }

    /* store the information */
    strcpy( new_element->name, name);

    new_element->size = size;

    for ( i=0; i<size; i++)
        new_element->blob[i] = blob[i];
    new_element->next = NULL;


    /* add the new element at the end of the list */
    if ( *list_pointer == NULL )
    {
        *list_pointer = new_element;
         new_element->next = NULL;
    }
    else
    {
        tmp_element = *list_pointer;

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

        tmp_element->next = new_element;
        new_element->next = NULL;
    }

    return 0;
}
```

```
/*
 *  Deletes the linked list defined by the pointer.
 *
 *  \param  **list_pointer Pointer to the linked list
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_worklist_destroylist"
void imgproc_worklist_destroylist( worklist_t **list_pointer)
{
    worklist_t *tmp_pointer;

    while( *list_pointer != NULL )
    {
         tmp_pointer  =  *list_pointer;
        *list_pointer = (*list_pointer)->next;

        /* delete name of the element */
        FREE(tmp_pointer->name);
        FREE(tmp_pointer->blob);

        /* delete element */
        FREE(tmp_pointer);
    }
    *list_pointer = NULL;
}


/*
 *  Deletes the element defined by the name.
 *
 *  \param   *name          Name of the element.
 *  \param  **list_pointer Pointer to the linked list
 *  \return                 0 on success otherwise -1
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_worklist_delete"
int imgproc_worklist_delete( char         *name,
                             worklist_t **list_pointer)
{
    worklist_t *tmp_pointer;
    worklist_t *del_element;
    int i=0;
    int position = 0;
    int found = 0;

    tmp_pointer = *list_pointer;

    /* find the  entry */
    while ( tmp_pointer != NULL )
    {
        if ( strcmp( tmp_pointer->name, name) == 0 )
        {
            found = 1;
            break;
        }
        tmp_pointer = tmp_pointer->next;
        position++;
    }

    if (found == 1)
    {
        /* it it is the first entry in the list, change the pointer to the
           linked list */
```

```
        if ( position == 0 )
        {
            tmp_pointer = *list_pointer;
            *list_pointer = (*list_pointer)->next;

            /* delete name of the element */
            FREE(tmp_pointer->name);
            FREE(tmp_pointer->blob);

            /* delete element */
            FREE(tmp_pointer);
        }
        else
        {
            tmp_pointer = *list_pointer;

            for (i=0; i<position-1; i++)
                tmp_pointer = tmp_pointer->next;

            del_element = tmp_pointer->next;

            tmp_pointer->next = del_element->next;

            FREE(del_element->name);
            FREE(del_element->blob);
            FREE(del_element);
        }
    }
    return 0;
}


/*
 *  Prints the names of the list elements.
 *
 *  \param  *list_pointer Pointer to the linked list
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_worklist_printlist"
void imgproc_worklist_printlist( worklist_t *list_pointer)
{
    worklist_t *current_element;

    current_element = list_pointer;
    while ( current_element != NULL )
    {
        IMGPROC_PRINT((
            IMGPROC_INFO(name = %s\n), current_element->name))
        current_element = current_element->next;
    }
}


/*
 *  Finds the position of an entry.
 *
 *  \param  *char          Entry to find
 *  \param  *list_pointer  Pointer to the linked list
 *  \return                The position or otherwise -1
 */
#undef  __FUNC__
#define __FUNC__ "imgproc_worklist_find_elementposition"
int imgproc_worklist_find_elementposition( char       *entry,
```

```
                                          worklist_t *list_pointer)
{
    worklist_t *current_element;
    int position = 0;

    current_element = list_pointer;
    while ( current_element != NULL )
    {
        if ( strcmp( current_element->name, entry) == 0 )
            return position;
        current_element = current_element->next;
        position++;
    }

    return -1;
}


/****************************************************************************
 *
 * END OF FILE
 *
 ****************************************************************************/
```

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