

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

**Parallel Object Oriented
Scattering Analysis using a
Service Oriented Architecture**

By
M.E.E.J. Koonen

Supervisor:
Rudolf Mak (TU/e)

Eindhoven, June 2006

Abstract

Massively parallel programs for computationally demanding scientific problems tend to be bound to specific hardware and/or software environments. As a consequence these programs do not scale well and cannot be ported easily. On the other hand it is known that resources of PC's in a LAN are underutilised. In this thesis we investigate the possibility of running a parallel program that harvests idle resources of nodes in a standard local area network using a service oriented architecture.

As a feasibility study we initially developed an object oriented sequential model and program that support simulation of scattering experiments. The simulation of such experiments is computationally demanding. To speed up the calculations this object oriented sequential model has been embedded in a service oriented architecture that supports simulation of scattering experiments in parallel. The first aim of this architecture was to harvest idle resources of nodes in a network. The speedup of such architecture is considered of secondary importance but is desired as well.

Based on this service oriented architecture we developed an object oriented parallel program as a proof of concept. This program showed that it is feasible to use a service oriented architecture to harvest idle resources on a standard local area network and gain a speedup as well.

Acknowledgement

I would like to thank Rudolf Mak for his guidance, support and criticism and Michel Chaudron and Gerard Zwaan for taking place in my examination committee. Further I want to thank Dave Jansen for the numerous interesting discussions about subjects that came into focus, Richard Verhoeven who helped with technical issues, Harry Niellissen with regard to his proof reading of parts of this document and I would like to thank my wife, Esther Quaedackers, for her invaluable support.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Structure of this thesis	2
2	X-ray Scattering Analysis	3
2.1	Scattering Experiments	3
2.2	Simulation	4
2.3	Computational determination of X-ray spectra	4
2.4	The Structure Factor	5
2.5	Radial Distribution Function	5
3	The OOSA Model	6
3.1	Generic Experiment	6
3.2	Sample	7
3.3	Outcome	7
3.4	Experiments	8
3.4.1	An abstract experiment	9
3.4.2	Structure of calculations in an SFS experiment	10
3.4.3	Structure of calculations in the SAXS, WAXS and MRD experiments	11
3.5	The sequential program	11
3.6	Practical Issues	12
4	Service Oriented Architecture	13
4.1	Resources	13
4.2	Service Oriented architecture	13
4.3	UPnP	14
4.3.1	UPnP Control Point and UPnP Service Interaction	15
4.3.2	The Cyberlink UPnP Stack	15
4.3.3	Data Communication	15
4.4	Practical Issues	16
5	The PROOSA model and application	17
5.1	Use Cases	17
5.2	Introduction of Tasks	17
5.3	The PROOSA Service	19
5.3.1	Initialisation action	19
5.3.2	Simulation actions	19

5.3.3	Finalisation action	20
5.4	PROOSA Service State Diagram	20
5.5	PROOSA Service Implementation	21
5.6	PROOSA Service control	23
5.6.1	Load Balancing	24
5.6.2	PROOSA Service Control Implementation	25
6	Performance Measurements	29
6.1	Samples	29
6.2	Measurement structure	29
6.3	Timing analysis	31
7	Conclusions	34
7.1	Future Work	34
A	OOSA diagrams	37
A.1	OOSA Package Diagram	37

Chapter 1

Introduction

This thesis describes the research performed in the context of my graduation project to obtain a master's degree in Computer Science. This project has been performed at the Eindhoven University of Technology, Department of Mathematics and Computer Science, in the expertise area System Architecture and Networking.

1.1 Motivation

Massively parallel programs for computationally demanding scientific problems tend to be bound to specific hardware and/or software environments. As a consequence these programs do not scale well and cannot be ported easily. An example of such a program is PARSTRUCT developed by van Gassel [21]. This program has been developed for transputer hardware and cannot be executed on other computation networks. Extension of this program to support other functionality as well is difficult because of the monolithic design.

An approach to overcome the binding of parallel programs to a specific hardware and/or software environment is to use a Service Oriented Architecture, SOA for short, for such programs. Programs that are based on a SOA are implemented using a framework that supports SOA's. An example of such a framework is known as Grid-technology. This technology focuses on large-scale resource sharing and addresses issues with respect to flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions and resources [15]. Grid technology provides a vast amount of services such as directory services, co-allocation, scheduling and brokering services, monitoring and diagnostics services, data replication services, workload management systems and collaboration frameworks, software discovery services, community authorisation services, community accounting and payment services and collaboratory services. These infrastructure services require non-trivial configuration, administration and management. For instance a community authorisation service requires administration of users that may use services in a network. We considered these requirements too demanding for the system we wanted to have.

In this thesis we investigate the possibilities of peer-to-peer computing for computationally demanding programs. Research shows [19] that only an average of 12% of the resources of desktop computers in a standard local area network is utilised. Our first aim is to harvest the vast amount of available free resources provided by networks communicating using standard web protocols. Although ultimately we would like our programs to be fast as well, we consider this of secondary importance in this thesis.

The solution in this thesis that has been designed to harvest unused resources has been based on a Services Oriented Architecture. This architecture has been implemented using the Universal Plug and Play (UPnP) technology [9]. This technology provides a mechanism for service discovery, advertisement and allows ad-hoc usage of idle resources of multi platform desktop computers in a network without the need for extensive configuration, administration and management procedures.

As a vehicle we used scattering analysis experiments [21] that are simulated by a parallel program. A preliminary object-oriented sequential model that supports the simulation of scattering experiments has been proposed by Mak[12]. This sequential model served as a starting point and has been re-factored and extended. Later this sequential model served as a starting point for the development of a parallel model and an implementation that is a proof of concept.

1.2 Structure of this thesis

Chapter 3 describes the entities and issues that are important in the domain of scattering analysis. Chapter 4 describes the sequential object-oriented model that served as the starting point for the parallel model and implementation. Chapter 5 describes the distributed infrastructure that is used in the the parallel model. Chapter 6 gives a detailed description of this parallel model. Chapter 7 gives measurements of execution times of sequential and parallel experiment simulations, shows speed up numbers and discusses these results. Chapter 8 gives conclusions and further recommendations.

Chapter 2

X-ray Scattering Analysis

This chapter provides some background on X-ray scattering analysis. It introduces the fundamental notions necessary to understand the OOSA and PROOSA models described in later chapters.

2.1 Scattering Experiments

X-ray scattering is a widely used and well established method to gain insight in the structural properties of all kinds of substances. It has its origin in the study of the atomic structure of crystals, but can also be used to study the structure of large molecules like proteins and polymers or aggregates of such molecules. Hence applications of X-ray scattering can be found in fields like crystallography, metallurgy, chemistry, biology and pharmacy.

In an X-ray scattering experiment a sample of the material to be studied is irradiated by a well-defined monochromatic X-ray beam. The electrons that surround the atoms in the irradiated sample are the entities which physically interact with the incoming X-ray photons. This interaction results in the scattering of the incoming X-ray beam resulting in a diffraction pattern of which the intensity spectrum can be measured. This spectrum provides useful information about structural properties of the sample and the distances between the particles present in the sample. Figure 2.1 shows a schematic setup of an X-ray scattering experiment.

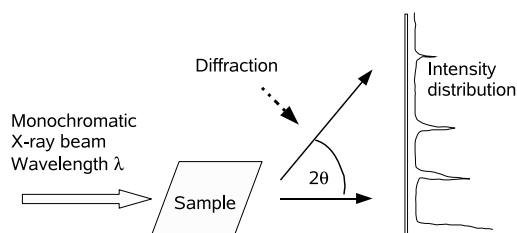


Figure 2.1: Schematic setup of a scattering experiment

2.2 Simulation

In a physical scattering experiment a spectrum is measured and the relative positions of the individual scatterers or derived quantities thereof are inferred. In a computer simulation of a scattering experiment the particle positions are given and the spectrum is derived. This enables researchers to compare spectra of experiments on unknown materials with calculated spectra of simulations. Sometimes it is useful to simulate such experiments on a computer if such experiments are physically impossible, e.g. if the time scale is too small or experiments are too costly. It may also happen that the sample itself is a result of another simulation, e.g. if the sample is a snapshot that is taken from another simulation. This situation occurs for instance in a molecular dynamics simulation.

2.3 Computational determination of X-ray spectra

Now that the purpose of the scattering experiment is clear, this section will discuss the calculation of intensity spectra. Let the scattered intensity be represented by function $I(q)$, where q is a function of the scattering angle. For models built of individual scatterers the formula of Debye [21] can be used:

$$I(q) = \sum_{j=1}^N \sum_{k=1}^N f_j(q) f_k(q) \frac{\sin(q r_{j,k})}{q r_{j,k}} \quad (2.1)$$

where:

q is the wave number. Given X-rays with wave length λ it is related to the scattering angle 2θ by the formula:

$$q = \frac{4\pi \sin\theta}{\lambda}$$

Hence the wave number is expressed in $1/\text{\AA}$.

N is the number of particles in the sample of the material.

$f_j(q)$ is the form factor of the j th particle. This factor expresses the scattering behaviour of the individual particles. In case a particle is an atom the form factor $f_j(q)$ is also known as the atomic scattering factor. For almost all atoms the atomic scattering factors, determined by initial calculations, are given in literature [22].

$r_{j,k}$ is the pair distance between particle j and particle k

In such intensity calculations we can distinguish several length-scales we are interested in. For instance there is the atomic length-scale within molecules or the length-scale within large aggregates of particles such as molecules in gels. For the measurement of these different length-scales a different wave-length for the X-ray beam is used.

When we are interested in small-length scales where crystal structure and the incorporated individual atoms play a role Wide Angle X-ray Scattering (WAXS) experiment is performed. For the larger-length scales where the structure of mesoscopic particles plays an important role a Small Angle X-ray Scattering (SAXS) experiment is performed. Small values for q in the calculation of $I(i)$ correspond to the SAXS part of the structure factor and large values correspond to the WAXS part.

2.4 The Structure Factor

Closely related to the intensity is a quantity called the structure factor. This structure factor abstracts from the identity of the individual particles and their individual scattering properties, retaining only structural information of the sample. Consider again Equation 2.1. Assuming that all particles are identical, there is the following relation between scattered intensity $I(q)$ and the structure factor $S(q)$:

$$I(q) = S(q) \cdot P(q) \quad (2.2)$$

where $P(q)$ is related to the form factor $f(q)$ by $P(q) = f^2(q)$. As stated the form factor of an individual particle captures the scattering behaviour of this particle. If we choose the form factor to be independent of the scattering angle then $f^2(q) = 1$ holds and we get:

$$S(q) = N + 2 \cdot \sum_{j=1}^{N-1} \sum_{k=j+1}^N \frac{\sin(q \cdot r_{j,k})}{q \cdot r_{j,k}} \quad (2.3)$$

In literature it is common to divide the formula for $S(q)$ by N :

$$S^*(q) = \frac{S(q)}{N} = 1 + \frac{2}{N} \cdot \sum_{j=1}^{N-1} \sum_{k=j+1}^N \frac{\sin(q \cdot r_{j,k})}{q \cdot r_{j,k}} \quad (2.4)$$

We now have the function $S^*(q)$ for the structure factor. Using 2.4 we can calculate the structure factor value for values of q .

2.5 Radial Distribution Function

The computation of $S^*(q)$ involves many summations of $\frac{N^2+N}{2}$ terms. To reduce the computational effort a histogram of pair distances can be compiled. When the number of entries in this histogram $\approx 10N$ good approximations of $S^*(q)$ can be obtained. Apart from a normalisation factor this histogram is nothing more but a discretised version of the radial distribution function $g(r)$. The radial distribution function (or RDF) is an example of a pair correlation function, which describes how, on average, the atoms in a system are radially packed around each other. This proves to be a particularly effective way of describing the average structure of disordered molecular systems such as liquids. Also in systems like liquids, where there is continual movement of the atoms and a single snapshot of the system shows only the instantaneous disorder, it is extremely useful to be able to deal with the average structure. Moreover, thermodynamical quantities like the energy of a system or its chemical potential can be related to the radial distribution function [13]. For these reasons the computation of a pair distance histogram is treated as a separate experiment in the next chapter, and not merely as part of the computation of intensity spectra or structure factors.

Chapter 3

The OOSA Model

Computer simulations that calculate structure factor spectra can be performed using the PARSTRUCT-program that has been developed by van Gassel [21]. The tight coupling to specific hardware and the monolithic design makes it difficult to extend this program to support the computation of other spectra. Hence it was necessary to develop a new object-oriented version. A preliminary sequential object-oriented model, which is called the object-oriented scattering analysis (OOSA) model, already existed [12]. This OOSA model supports the calculation of structure factors, SAXS spectra and WAXS spectra. The structure of the OOSA model makes it easy to add simulations of other experiments. Because the OOSA model is of importance to the parallel model introduced later, the structure and elements of the OOSA model are discussed in this chapter.

3.1 Generic Experiment

The OOSA model provides a framework for the simulation of various scattering experiments such as SFS, SAXS and WAXS-experiments. The design of this framework has been based upon a generic view on experiments [16]. In this view an experiment is conducted on some subject which results in measurements of some quantities. In OOSA a complete collection of measurements is referred to as the outcome of an experiment where an individual measurement is called an observation. For example for an SFS experiment, the outcome of this experiment is the set of calculated structure factors and an individual calculated structure factor is an individual observation.

In case of scattering experiments the subject is a sample of the substance or mixture of substances that is irradiated by a beam of X-rays. The quantity that is measured in such experiments is the intensity of the scattered X-ray beam as a function of the wave number. The diagram in Figure 3.1 shows the relation between these elements as incorporated the OOSA model. The diagram in Figure 3.1 makes clear that an experiment is conducted on a sample which represents the subject. An Observation object, observation for short, represents a single measurement. It specifies the values of all quantities involved in that measurement given in the appropriate units. An Outcome object, outcome for short, is an aggregate of all measurements performed in an experiment. This generic representation of an experiment allows to incorporate other experiments besides scattering experiments as well. As an example we could define a ray tracing experiment where a scene graph is the sample and the calculated pixel colour matrix is the outcome. The elements of the diagram in Figure 3.1 are discussed in detail in the next section.

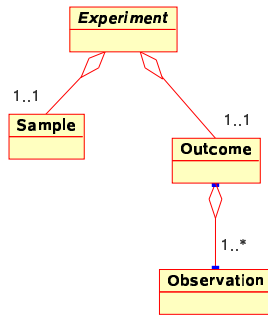


Figure 3.1: Conceptual diagram of an experiment

3.2 Sample

In the context of scattering experiments a sample consists of a three-dimensional box containing particles. These particles are the individual scatterers in scattering experiments. Every particle in this box is located at a position. The set of all particles can be divided into sub sets called substances whose members are considered identical with regard to a specific experiment. In a scattering experiment particles of the same substance have identical scattering behaviour. This scattering behaviour is expressed using the form factor as discussed in the previous chapter. A sample is called homogeneous if its particles belong to a single substance, otherwise it is called heterogeneous. Figure 3.2 shows the class diagram with respect to a sample in OOSA as described in this section.

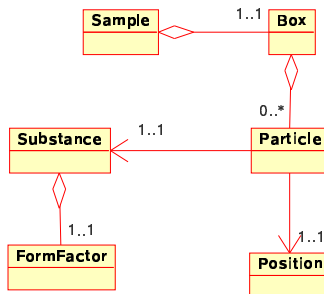


Figure 3.2: Class diagram w.r.t a sample

3.3 Outcome

The entire collection of measurements for an experiment simulation is represented by the class Outcome as described in the class diagram in Figure 3.3. Such an outcome consists of individual measurements that are called observations. An observation is modelled by class Observation and contains a finite non-empty sequence of Quantity objects. Quantity objects model physical quantities which are values with corresponding units which are modelled by class Unit. The International System of Units (SI) specifies [3] the units in which each (physical) quantity is measured. Each unit has

both a name and a symbol. A Unit-object stores both the name and the symbol of a unit.

Some quantities assume values from a discrete domain, like amount of substance quantities, whereas other quantities assume values from a continuous domain, like length quantities. Sampled quantities are quantities whose values are taken from a discrete ordered set. Each quantity has a signature that consists of a name, which uniquely identifies the quantity, the type of the values the quantity can assume, and a lists of units in which these values are expressed. E.g. the quantity named speed is expressed in m/s and its value can be an arbitrary non-negative real value.

If the sequence of quantities in an observation has only one element, then this a Quantity object. If the sequence has two or more elements, then all but the last element must be SampledQuantity objects and the last one must be a quantity object. The former observations are used to model constants, like for instance the wave length of the X-ray source used in a SAXS experiment. The latter observations are used to model physical entities that are functions, such as for instance the intensity in a SAXS experiment. So the initial subsequence of sampled quantities represents a particular point in the domain of the function and the final quantity object represents the corresponding result. Constants can be viewed as zero argument functions, thus observations always used to model functions. Of course many observations are needed to represent a function with some accuracy.

Just like a (sampled) quantity has a signature and a value, so has an observation. The value of an observation is called a signal modelled by class Signal. An object of this class consists of a value for each of the arguments and a value for the result.

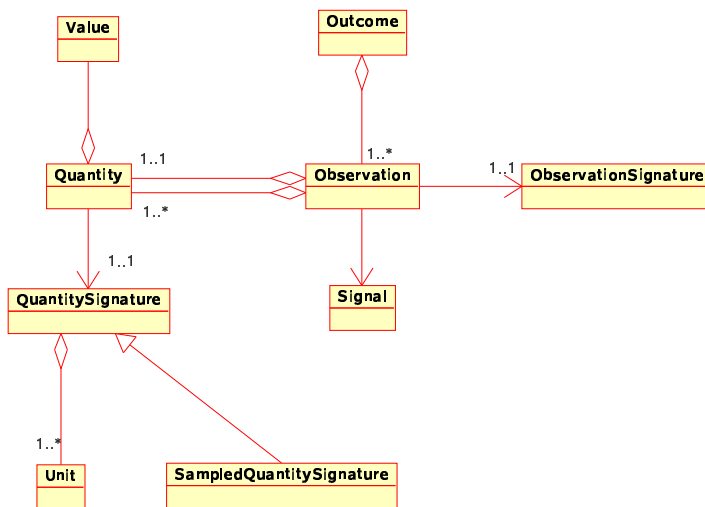


Figure 3.3: Class diagram w.r.t outcome

3.4 Experiments

The experiments that are supported by the OOSA-model are shown in Figure 3.4. This figure shows the abstract base class Experiment together with the classes that model the experiments that are available. For the SFS, SAXS and WAXS experiments the corresponding spectrum is calculated. The purpose of SRD (Single Radial Distribution) and MRD (Multiple Radial Distribution) experiments is discussed later.

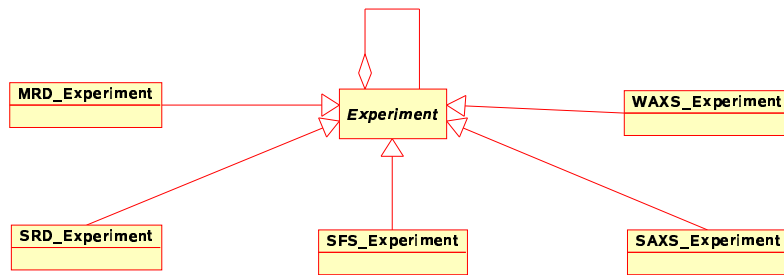


Figure 3.4: Class diagram of experiments available in the OOSA-model

3.4.1 An abstract experiment

The design of experiment classes in OOSA follows the *Template Method* design pattern which defines the skeleton of an algorithm in an operation, deferring some steps to subclasses [17]. Using this pattern, subclasses can redefine certain steps of an algorithm without changing the algorithm’s structure. In the OOSA-model, the base class *Experiment* defines this algorithm for the simulation of experiments by the method *simulate* which defines the steps of the algorithm and thus defines the canonical behaviour of an experiment. One of these steps is, e.g. a repeated invocation of an abstract method *performObservation* that is implemented in derived classes. Figure 3.4 shows the class interface of class *Experiment* and Figure 3.6 shows the sequence diagram that captures the algorithmic structure of Template Method *simulate*.

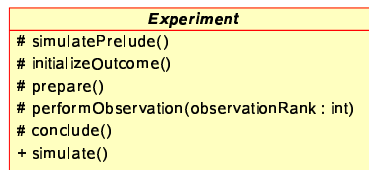


Figure 3.5: Class diagram: canonical Experiment interface

The execution of a simulation of an experiment results in an outcome that has been calculated. Now we discuss the generic algorithm that is used, describing the steps that are involved. When we refer to methods these are the methods of base class *Experiment* in Figure 3.5. In the first step a prelude experiment is simulated if its outcome is needed. A prelude is an experiment of which the outcome is needed for the simulation of the main experiment. The simulation of such a prelude experiment step is represented by method *simulatePrelude*. An example of a prelude experiment will be given in the next section. After the execution of the prelude experiment the outcome structure of the main experiment is initialised. This step is captured in method *initializeOutcome*. The following step where experiments can implement additional actions if needed is captured by method *prepare*. After the prepare step the actual observations are performed by a repeated invocation *performObservation*. The outcome that is calculated may need some post processing which can be performed in method *conclude*.

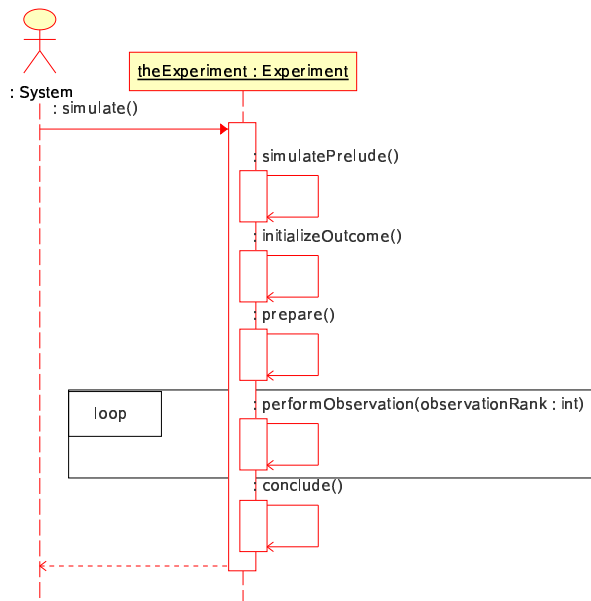


Figure 3.6: Sequence diagram of the canonical behaviour of an experiment

3.4.2 Structure of calculations in an SFS experiment

Now that we have defined the interface of a generic experiment and the template method *simulate* we can describe the structure and behaviour of the structure factor spectrum (SFS) experiment. Figure 3.7 shows that class `SFS_Experiment` overrides two protected methods of base class `Experiment`, i.e. the methods *prepare* and *performObservation*. First we will discuss the method *performObservation*.

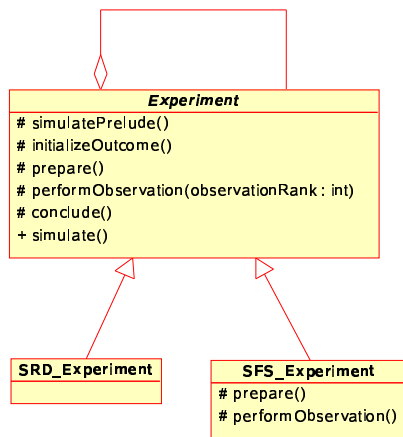


Figure 3.7: Base class `Experiment` and sub class `SFS_Experiment`

The *performObservation* method

The method *performObservation* of class `SFS_Experiment` provides the implementation for the calculation of the structure factor for a value of q . This structure factor could be calculated by using Equation 3.1

$$S^*(q) = 1 + \frac{2}{N} \cdot \sum_{j=1}^{N-1} \sum_{k=j+1}^N \frac{\sin(q \cdot r_{j,k})}{q \cdot r_{j,k}} \quad (3.1)$$

but it would be very inefficient to calculate all pair distances for every value of q . Pre-calculation and storage of all distances is more efficient since these values can be reused. Because of the large amount of distances and limited storage capacity in general, not all of these pair distances can be stored. Therefore we introduce a pair-distance histogram that is calculated by the single radial distribution (SRD) experiment before the simulation of an SFS experiment. This SRD experiment now is a prelude experiment to the SFS experiment and will be simulated before the simulation of the SFS experiment. In such an SRD experiment all pair distances that are present within a specified distance-range are counted and stored in the outcome. The size of a distance-range is defined by the bin-size. Note that in an SRD experiment the substance of the particles does not play any role. The introduction of the SRD experiment that calculates histogram M gives the following equation for the calculation of the structure factor:

$$S^*(q) = 1 + \frac{2}{N} \cdot \sum_{i=1}^{N_{bins}} M(r_i) \cdot \frac{\sin(q \cdot r_i)}{q \cdot r_i} \quad (3.2)$$

where $M(r_i)$ is the bin population at pair distance r_i and N_{bins} is the number of bins in the histogram M .

The *prepare* method

In the method *prepare* the outcome that is generated by the SRD experiment is processed. All empty bins in the histogram are removed to speed up the calculation in the *performObservation* stage.

3.4.3 Structure of calculations in the SAXS, WAXS and MRD experiments

The calculation of the SAXS and WAXS spectra are similar to the calculation of the SFS spectrum. The difference is that the form factor of the particles in the sample play a role. The histogram that is created by the Multiple Radial Distribution (MRD) experiment stores pair distances together with the substance of the particles. The type of the particles is of importance since the form factor of the particles is needed in the SAXS and WAXS experiments according to Equation 2.1.

3.5 The sequential program

The sequential program for the simulation of experiments has been developed using the classes in the OOSA model. This section discusses the structure of this application in a short and operational way.

The execution of the application can be divided in several phases. These phases are shown in the state diagram in Figure 3.8. First the application reads in a definition file and a sample file that are provided by the user. In the definition file a user defines simulation parameters, e.g. the range of the wave number for which the intensity has

to be computed. The sample file contains a description of the sample that is used in the experiments. It describes the particles, their positions and their substance type. The application uses both files to construct the objects needed for the simulation of the experiments. In the simulation phase the actual calculations of experiments take place. Results of these calculations, i.e. the outcomes, can either be stored to disk or reused by consecutive experiments. In the finalisation phase all objects are destructed and the application ends its execution.

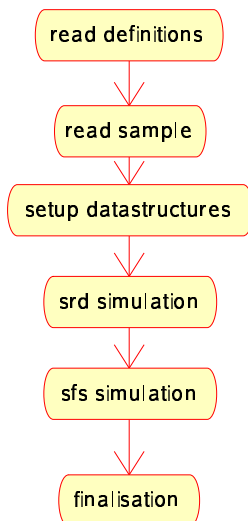


Figure 3.8: Phases in the sequential application

3.6 Practical Issues

As indicated the current sequential model and program were not developed from scratch. First the complexity of the preliminary model was reduced by removal of several Factory classes. All classes with regard to two dimensional vectors have been removed. This reduction of classes yielded a much simpler model. The remaining classes have been grouped into packages and sub packages based on the purpose of the classes within the model. This yielded a clear package structure with packages that contain closely related classes. To enable easy loading and creation of Sample objects a sample parser class has been added. Class SRD_Experiment has been extended such that it supports the calculation of a *partial* pair-distance histogram. This extension consisted of the introduction of two extra parameters that specify the set of particle pairs of which the distances have to be calculated. Class Outcome has been extended with a combine method that reads in a partial outcome from file and adds the values to its own already existing histogram. Although of usage in the sequential case this functionality is mainly used in the parallel program.

On the source code level each class definition and class implementation now has its own file. These files have been put into a directory structure that reflects the package structure mentioned previously. This structure provides an implementation package structure similar to the logical package structure and makes the implementation code easy to understand. For easy compilation of this source code a makefile structure has been made. Extension of this makefile structure is straightforward and makes addition of new source code for classes easy.

Chapter 4

Service Oriented Architecture

Simulation of a scattering experiment is a computationally demanding task. The execution time of such a sequential simulation using the OOSA model can be shortened by executing parts of these simulations in parallel. This requires a system that enables parallel execution. This chapter describes the Service Oriented Architecture that has been used to embed the OOSA model into a parallel architecture called PROOSA.

4.1 Resources

It is a fact that resources of many computer networks are underutilised, but hard numbers are difficult to obtain. Research shows that the resource usage of available resources is in average below 12% [19]. Given the fact that a large number of such networks exists within universities, companies and government institutions, a vast amount of unused resources is present. These unused resources can be deployed for other purposes such as parallel simulation of scattering experiments.

4.2 Service Oriented architecture

In this thesis we develop a Service Oriented Architecture to enable usage of otherwise unused resources on nodes in a local area network. Such a Service Oriented Architecture, SOA for short, is a collection of communicating services which use message passing as their means of communication. The communication can be between two services, between a service-user and a service or it can be multiple services coordinating an activity [14]. A service in a service oriented architecture is a contractually specified functionality and a service has an interface, specification, quality of service and an access point.

The functionality of a service is made available to a service user through the service interface in the form of actions, responses and events. The specification of a service describes its functional properties, i.e. the effect of an action on the output parameters and the state variables of a service. The specification also describes the behavioural properties of a service, i.e. how and when actions should be called and how responses are generated. Additional to the specification there is a specification of the quality of the service, for example in terms of performance or reliability. The service access point provides the actual access to the functionality of a service. The specified functionality

of a service can be implemented using one or more objects, components or what is appropriate for the implementation.

Since we intend to use services in a Service Oriented Architecture using a local area network for service deployment and communication we need additional properties to a service as specified above. A service must be accessible over the network otherwise it will not be usable for other users in the network. Further a service must have the ability to advertise itself on the network such that other users may detect its presence. On the other hand, service users need to be able to dynamically discover services on the network.

The idea now is that a user runs a service on her own computer in the network without considerable configuration and administration issues. This computer provides unused resources to other users on the network by means of the specified functionality of the service running on that computer. Other users are able to discover and use this functionality by calling the actions of the service. In our architecture the services we use are not unlike standard services. Standard services provide just one task or function such as switching a light. Our service provides all functionality that may help service users in parallel simulation of scattering experiments.

4.3 UPnP

Several technologies for the implementation of service oriented architectures exist. Grid technology provides a vast amount of services such as directory services, co-allocation, scheduling and brokering services, monitoring and diagnostics services, data replication services, workload management systems and collaboration frameworks, software discovery services, community authorisation services, community accounting and payment services and collaboratory services [15]. These infrastructure services require non-trivial configuration, administration and management. We consider this technology too heavy weight to use for our purposes.

A more light weight technology that exists is the JXTA technology. This technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner [7]. JXTA peers create a virtual network where any peer can interact with other peers and resources directly. This technology is platform and programming language independent but it has its focus on Java. This makes usage of the JXTA technology within the OOSA code that is written in C++ not that straightforward.

Our parallel implementation for simulation of scattering experiments uses the Universal Plug and Play protocol, UPnP protocol for short, as protocol for the discovery, advertisement and control of services. We use this protocol because it can be used easily within the existing C++ code base. UPnP is a light weight protocol and there is already knowledge and practical experience present within SAN [14]. The UPnP architecture [9] is a distributed, open networking architecture that uses Internet-based technologies such as TCP/IP, UDP, HTTP and XML. UPnP has its origin in the area of consumer devices such as routers and printers but it is media, device, operating system and programming language independent. In the UPnP protocol there are UPnP devices which aggregate one or more UPnP services. These services provide functionality, e.g. dimming or switching a light [10], where a device is an access point to a service or a set of services. The functionality of a service is specified by an interface. This interface specifies the actions that can be invoked, their parameters and responses and the possible events that may occur. Actions can be invoked by a UPnP control point that has discovered a service.

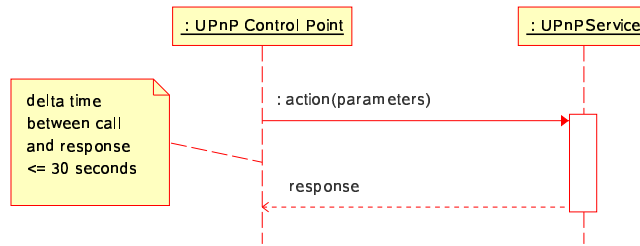


Figure 4.1: Time span between synchronous action call and its response should be less than 30 seconds

4.3.1 UPnP Control Point and UPnP Service Interaction

The discovery of UPnP devices and services in a network is automatically taken care of by a UPnP control point. In case a UPnP device has been discovered by a UPnP control point, the control point adds this device to its set of existing UPnP devices automatically. In the situation that a device leaves the network this device is automatically removed from this set. The control point may query devices for services and invoke actions of these services. The invocation of an action of a service and its response must follow the UPnP protocol. According to this protocol the time span between a call of an action of a UPnP service and its response may last at most 30 seconds [14], as shown in the sequence diagram in Figure 4.1.

For PROOSA we want to be able to invoke actions of services in parallel without the introduction of threads in the control point. Therefore we transform a synchronous action invocation of a service into an asynchronous action by the following pattern: instead of performing an action directly after invocation, the action logic is performed in a thread which raises an event after completion. Now the direct response of the action indicates whether the action thread has been started successfully or not. In case a thread ends its execution it generates an event. This event contains the eventual execution status of the thread. Reception of an event by a control point implies that the corresponding action has been completed. Figure 4.2 shows a sequence diagram that illustrates such an asynchronous pattern.

4.3.2 The Cyberlink UPnP Stack

For the actual implementation of UPnP control points, devices and services several UPnP stacks exist [1]. For our system we use the “Cyberlink for C++” UPnP stack because the OOSA classes have already been implemented using C++. This UPnP stack is freely available open source software and can be downloaded on the Internet [2]. “Cyberlink for C++” provides the base classes for a UPnP control point and devices and provides functionality for services which can be extended to fit the needs of the developer.

4.3.3 Data Communication

Until now we have only discussed the service control structure. However, as we will see in next chapter, data must be communicated between services or between services and the control point. The UPnP protocol does not provide an efficient mechanism to transfer large amounts of data. Therefore it is decided to use the File Transfer Protocol [11] for the transfer of data. This decision has implications for the implementation in the next chapter: services need to be able to retrieve files using FTP client functions. This is implemented using FTP client classes of the Trolltech Qt library [8]

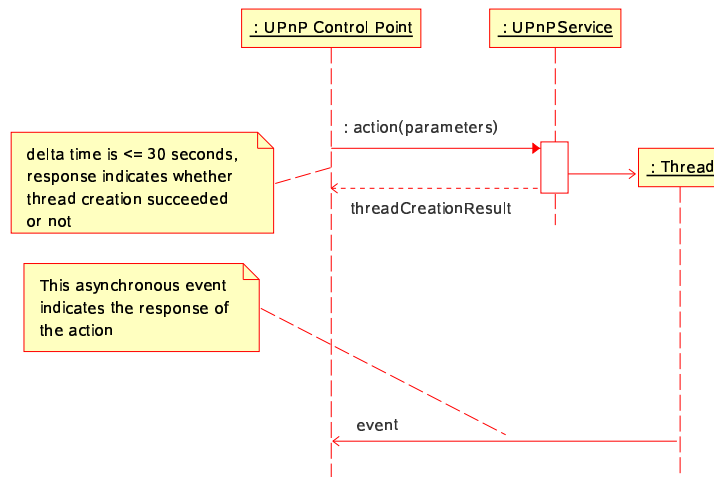


Figure 4.2: Introduction of asynchronous action call

in the implementation of the actions of services. A service must be accompanied by FTP server functionality that provides access to the files of this particular service. In this thesis the FTP server functionality is implemented by usage of an open source FTP daemon called “Indiftpd” [4]. It is a fact that the FTP protocol is not secure but in this thesis it is only used as a proof of concept. A secure solution would be the Secure FTP protocol [6], but this is beyond the scope of this thesis.

4.4 Practical Issues

Unused resources of nodes in a network will be made available by running a local script that starts an application that implements a UPnP device with a UPnP service together with an FTP server daemon. Both the application and the daemon are started with a low priority. The owner of the machine decides when to start the script. Therefore it is only necessary that the ports used by the device, service and FTP server are not blocked by a firewall. Another user can, by a running application that uses or implements a control point, search for devices with a specific service and use the actions specified in the interface of this service. If an owner of a node does not want to contribute the node to the network anymore they only have to shutdown the service and FTP server using a single script.

Chapter 5

The PROOSA model and application

In the previous chapter we discussed an Universal Plug and Play based Service Oriented Architecture. In this chapter we show how this architecture is used for parallel simulation of scattering experiments. First the entire simulation is partitioned into several tasks. These tasks are executed using multiple instances of a PROOSA service distributed over a network. Each action of such a PROOSA service performs a task. The interface of the PROOSA Service that is used for calculations will be introduced and discussed. Instances of this PROOSA service are used for simulations by invoking actions of these services. The part of the system that is responsible for this invocation and the coordination of invocation of these actions is performed by the PROOSA Service Control. This service control is also discussed.

5.1 Use Cases

The use case diagram for the parallel system that has been developed is rather simple. It indicates that a user that wants to simulate experiments must provide the following input:

- The location of the file that contains execution parameters
- The location of the file that contains the sample definition that is used during the simulations
- The number of nodes (services) that the user wants to use for the execution of the simulations. In case this number is not specified the system will use the number of services that are available.

Figure 5.1 shows the use case diagram.

5.2 Introduction of Tasks

Based on the use case diagram in Figure 5.1 we consider the tasks:

- The processing of a definitions file
- The processing of a sample file
- The processing of the number of nodes to be used for the calculation
- The execution of simulations

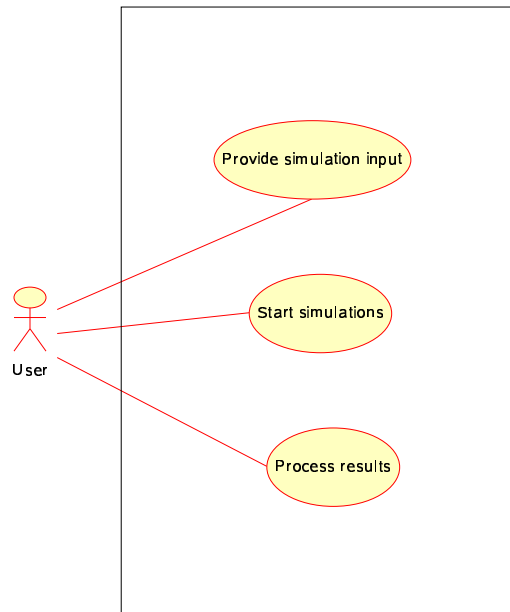


Figure 5.1: Use case diagram for the parallel system

1. Simulation of the Single Radial Distribution Experiment
2. Simulation of the Structure Factor Spectrum Experiment

We only consider the simulation of Single Radial Distribution and Structure Factor Spectrum experiments. Multiple Radial Distribution and Small and Wide Angle Scattering experiments follow the same pattern.

All experiments are performed in a Service Oriented Architecture using the Master-Worker pattern [20]. The application that the user starts is the master that controls a single or multiple instances of the PROOSA service in a network. These PROOSA services are the workers that simulate parts of experiments. First we will introduce the interface of the service and the design of the system triggered by an invocation of an action in this interface. Then we will show how actions of these services are used by the master.

Since the simulation of an Single Radial Distribution experiment is computationally demanding for samples containing a large number of particles this task is refined into the following sub tasks:

- tasks that calculate a partial outcome of an Single Radial Distribution experiment
- tasks that combine two partial outcomes

Each task that simulates a Single Radial Distribution experiment consists of a set of concurrent subtasks that calculate a partial pair-distance histogram. After completion a set of *combine* subtasks take these partial outcomes and combine them to the total histogram that is the outcome for an Single Radial Distribution experiment. The workload for an SFS experiment is that small therefore the simulation of an SFS experiment is not divided into partial tasks. One service simulates an entire SFS experiment.

5.3 The PROOSA Service

To be able to work on the tasks defined in the previous section a PROOSA service needs to provide actions that can be used by master to execute these tasks. Therefore the PROOSA service interface defines 5 actions which will be discussed in detail in the following subsections:

- *initialiseService*
- *simulatePartialSRDExperiment*
- *combinePartialSRDOOutcome*
- *simulateSFSExperiment*
- *finaliseService*

All of the listed actions create a thread and their direct response is the creation status of the thread. This thread implements the functionality of the corresponding action. Each thread sends, after completion, an event that indicates whether the thread completed successfully or not. Each service will only work on one action a time.

In the definition of a service it is stated that the specification of a service interface also contains information about Quality of Service properties. Here we will not consider these properties.

5.3.1 Initialisation action

Action *initialiseService* performs the initialisation of the service. This action fetches the files specified in the parameter list and initialises the object that are necessary for the simulation of experiments.

```
bool initialiseService( url definitionFile,
                      url sampleFile )
pre   : service has not already been initialised and
        definitionFile url is valid and
        sampleFile url is valid
post  : definitionFile and sampleFile have been fetched and stored
        and the service has been initialised
event : completion notification
```

5.3.2 Simulation actions

Action *simulatePartialSRDExperiment* generates a partial histogram of pair distances for the particles in the sample for the sub set

$$[particleStartRank, \dots, particleStartRank + particleCount - 1]$$

Thus if $particleStartRank = 1$ and $particleCount = \text{total number of particles in a sample}$ the entire histogram is calculated. The outcome of this calculation is stored in a file.

```
bool simulatePartialSRDExperiment( int particleStartRank,
                                  int particleCount )
pre   : service has been initialised
        1 <= particleStartRank <= #particles in sample
        1 <= particleCount <= #particles in sample
        particleStartRank + (particleCount-1) <= #particles in sample
post  : service has calculated and stored a histogram of
        pair distances for all particle pairs (p1,p2) where
```

```

    p1,p2 in
    [particleStartRank, .. , particleStartRank + particleCount-1]
    This histogram is stored in a file on the service. The name of
    this file is constructed out of the system name and extension
    '.srd'. The system name is provided by the definition file.
    event : completion notification

```

Partial SRD outcomes can be combined using the action *combinePartialSRDOutcome*. Action *combinePartialSRDOutcome* fetches the file specified by the URL and constructs a combined histogram out of its own existing outcome and the outcome in the file. This combined outcome is stored to a file. Action *combinePartialSRDOutcome* can be invoked multiple times.

```

bool combinePartialSRDOutcome( url partialSRDOutcomeFile )
pre  : partialSRDOutcomeFile url is valid
post : the contents of the partialSRDOutcomeFile has been combined
      : to a new histogram. This histogram is stored in a file on
      : the service. The name of this file is constructed out of the
      : system name and extension '.srd'. The system name is provided
      : by the definition file.
event : completion notification

```

Action *simulateSFSExperiment* calculates the Structure Factor Spectrum for an SRD outcome.

```

bool simulateSFSExperiment( url srdOutcomeFile )
pre  : url SRD outcome is valid
post : the srdOutcome had been fetched and stored and
      : the SFS spectrum has been calculated and stored in a file on
      : the service. The name of this file is constructed out of the
      : system name and extension '.sfs'. The system name is provided
      : by the definition file.
event : completion notification

```

5.3.3 Finalisation action

Finalisation of a service is performed by the action

```

bool finaliseService()
pre  : service must at least be initialised
post : all internal structures have been finalised
event : completion notification

```

Action *finaliseService* destructs all structures necessary for the simulation of experiments. Now a service can be initialised again for usage.

5.4 PROOSA Service State Diagram

The invocation and order of invocation of actions of a PROOSA service that have been discussed are restricted to the states that are shown in the state transition diagram in Figure 5.2. For the sake of clarity this diagram does not show that, once a service has been initialised, it is possible to finalise it, no matter what state the service is in.

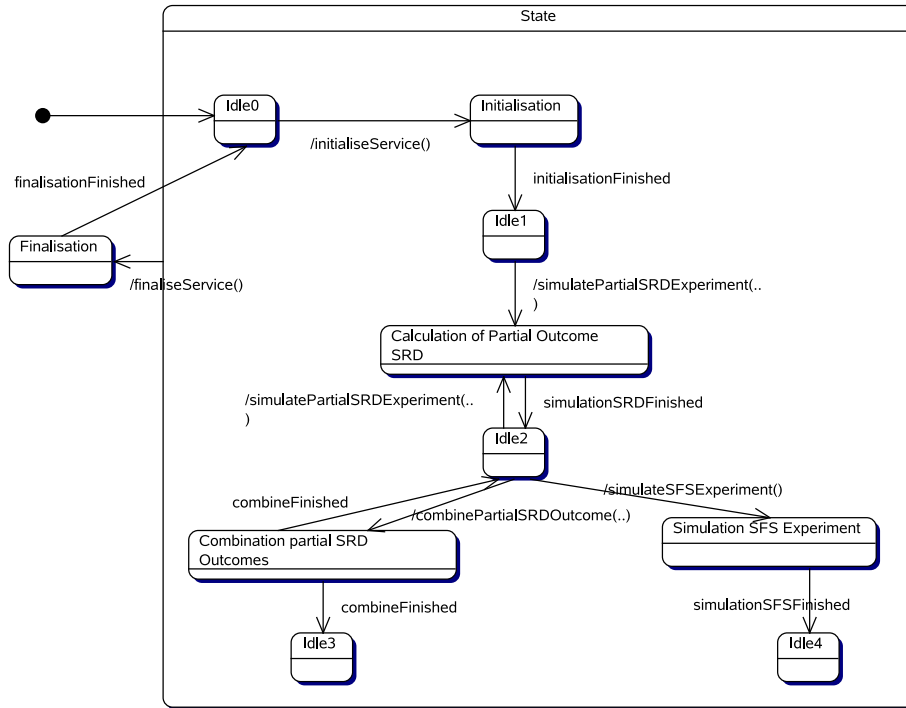


Figure 5.2: State Transition Diagram for a PROOSA service

5.5 PROOSA Service Implementation

The actual UPnP implementation of the PROOSA Service with UPnP devices and services using the aforementioned “Cyberlink for C++” UPnP stack is rather straightforward. The interface of the PROOSA service need to be specified using an XML file containing the actions, action names, parameters, etc. A UPnP device aggregates the PROOSA service. For this UPnP device an XML file must be specified that contains the definitions for the device type, manufacturer, etc. and a list of services that are aggregated by the device. In the current implementation a device contains only one service.

The UPnP device serves as an access point to a service and its actions and advertises its services, as defined in the XML file, to the network. A service user can query a control point for available services and the interfaces of these services. Invocations of actions specified by these interfaces are managed by the UPnP device. Such device forwards the actions to the actual implementation of an action. As stated the actual work of an action is performed by a thread created by the POSIX thread library [5].

The completion notification event extension as discussed in the previous chapter has been implemented using state variables of a UPnP service. Once a service user, i.e. a UPnP control point, has been subscribed to such a state variable it receives events if the value of such a variable changes. The value of such events is the execution state of the action that generated the event, i.e. an event has the value “OK” or “NOTOK”. For each action in the PROOSA service there is a corresponding state variable. Therefore

a service has the following state variables:

- initialiseServiceDone
- simulatePartialSRDExperimentDone
- combinePartialSRDOutcomeDone
- simulateSFSExperimentDone
- finaliseServiceDone

which are all booleans representing whether the action completed successfully or not.

Execution of (the thread in) action *initialiseService* takes care of the initialisation of the PROOSA service. The underlying implementation uses the classes of the OOSA model. In the initialisation phase objects of classes *Sample*, *Outcome*, *SRD_Experiment* and *SFS_Experiment* are instantiated. Action *simulatePartialSRD_Experiment* uses an *SRD_Experiment* object for the calculation of the partial outcome. This object has properties that indicate the set of particles in the sample the of which the pair-distances must be counted. Action *combinePartialSRDOutcome* uses an *SFS_Experiment* object for the calculation of the Structure Factor Spectrum. Action *finaliseService* takes care of the destruction of the objects instantiated in the previous actions.

Figure 5.3 illustrates the most important classes and parts of the interfaces essential for the understanding of the implementation. Class *Device* and *ActionListener* are two

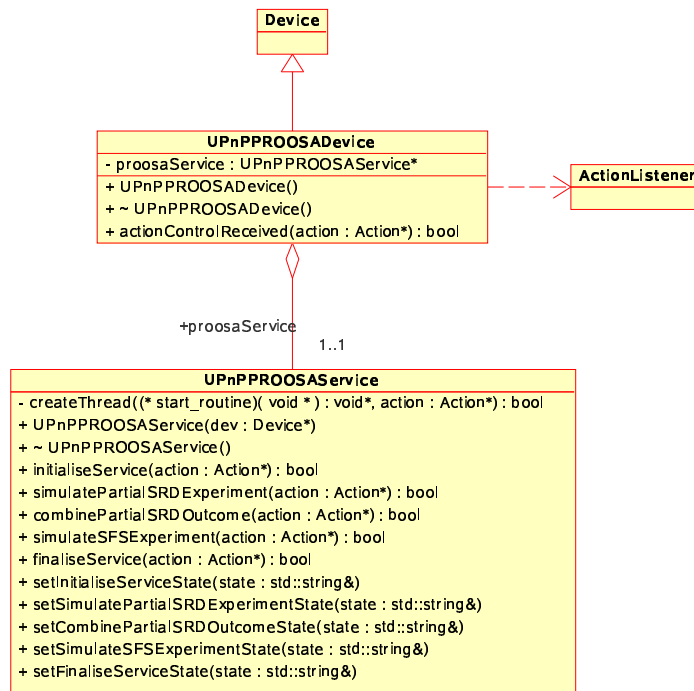


Figure 5.3: Conceptual class diagram of a PROOSA Device and Service implementation

classes that are provided by the “Cyberlink for C++” UPnP stack. Class *Device* has basic functionality with regard to advertisement, the loading of the XML definition file etc. Class *UPnPPROOSADevice* derives from *Device* and implements the access to the actions of a *UPnPPROOSAService*. Invocation of actions is handled by an

ActionListener. Method *actionControlReveiced* of UPnPPROOSADevice is called if the device receives an action call by a service user, i.e. a UPnP control point. By usage of an ActionListener in this method the action is analysed and the corresponding method of the UPnPPROOSAService is called. For instance, when a service user calls action *initialiseService* of the service interface, the method *initialiseService* of UPnPPROOSAService is called.

Class Action that is used in the interfaces of UPnPPROOSADevice and UPnPPROOSAService is a class provided by “Cyberlink for C++” that models the administration with regard to action calls. It contains the name of the action, the parameter list etc.

5.6 PROOSA Service control

The general idea for the PROOSA service control (the application that controls the services) is that it is the master in the master-worker pattern that initiates the simulation. The PROOSA service control creates a *set of tasks* and waits until the job is done, consumes the results and then shuts down the computation [20]. In our specific situation several sets of tasks are created:

1. a bag of initialisation tasks
2. a bag of partial SRD simulation tasks
3. a bag of combination tasks
4. a bag of a single SFS simulation task
5. a bag of finalisation tasks

which correspond with the action names in the PROOSA services. The tasks in 1), 2) and 5) are performed for each service, where the tasks in 4) and 5) are performed on a sub set of the total set of services that are available.

A straightforward, too strictly synchronised implementation for the master in case a Single Radial Distribution experiment and Structure Factor Spectrum experiment must be simulated would be the following:

- wait for the desired amount of services on the network
- perform all initialisation tasks on all services
- wait until all services are ready
- perform partial SRD simulation tasks on all services
- wait until all services are ready
- perform all combination tasks on selected services
- wait until all services are ready
- perform SFS tasks on one selected service
- wait until service is ready
- fetch results
- perform finalisation tasks on all services

As stated before the synchronisation in this schedule is too strict. First, if a service has been initialised it is not necessary to wait for this service since the partial simulation of an SRD experiment is not dependent on other data or service. Therefore, after the reception of the completion event of an *initialiseService* action of a worker, the master can directly order the execution of a partial Single Radial Distribution simulation on the same service. Combination of partial SRD outcomes is already possible if there are at least two services that have finished their partial SRD simulation and hence

are also available for combination. In this situation the master can also schedule the combination tasks earlier than specified before. The only strict synchronisation points are forced by the pre-condition of the SFS simulation and the pre-condition for the fetch of results.

The SFS simulation requires a complete SRD outcome for the calculation of the structure factor spectrum. For performance reasons it is desirable that this SRD outcome is already on the service that performs the SFS experiment. The result of the SFS experiment can only be fetched after completion of this experiment. Figure 5.4 shows this in a state transition diagram for the master with concurrent states with regard to service n , where $1 \leq n \leq N$ and N is the cardinality of the set of PROOSA services that are available to the master.

5.6.1 Load Balancing

Now we discuss the load balancing of the simulation of an SRD experiment. First we consider the total workload for an SRD experiment. This workload is the number of pair-distances that must be counted for the construction of the histogram. The workload is determined by the number of particles in the sample. The total load TL for the calculation of a total histogram is

$$TL = \frac{N^2 + N}{2}$$

where N is the number of particles in the sample. The smallest task size defined for an SRD experiment is the construction of a partial pair-distance histogram for distances of pairs in the set:

$$T_{p_a}\{(p_a, p_j) | a \leq j \leq N\}$$

where p_a is a particle with rank a , $1 \leq a \leq N$, and p_j is a particle with rank j . The load of such a task T_{p_a} is given by

$$-a + N + 1$$

The total load for an entire histogram then is

$$\langle \sum_i : 1 \leq i \leq N : -i + N + 1 \rangle$$

Parallel calculation of N particle ranks by N services is heavily unbalanced so we decide to aggregate fine grained tasks. Let S be the set of available PROOSA services and $M = |S|$. We want to distribute the total calculational load over all services and have services to perform a computation with a partial work load PL

$$PL = \frac{TL}{M}$$

However this is not possible because of the inherent imbalance of the load for a fine grained task T_{p_a} . Therefore we have a weaker constraint for the individual load l of a service: $|l - PL| \leq R$. A task for service i is given by a set of particle rank pairs:

$$T(a_i, b_i) = \{(k, l) | a_i \leq k \leq b_i \wedge k \leq l \leq N\}$$

where

- a_i and b_i are particle ranks were $1 \leq i \leq M$
- $a_i \leq b_i \leq N$
- $a_1 = 1$ and
- $b_M = N$

The set of tasks for all services is now constructed such that

$$\langle \forall_i : 1 \leq i \leq M : |load(a_i, b_i) - PL| \leq R \rangle$$

where

$$\langle \Sigma_i : 1 \leq i \leq M : load(a_i, b_i) \rangle = \frac{N^2 + N}{2}$$

An Example

As an example we consider a set of tasks for $N = 20$ and $|S| = M = 4$ and $R = 5$. The total load for this example is

$$\frac{N^2 + N}{2} = \frac{420}{2} = 210$$

and the ideal load per service then is 52.5. The set of tasks that is constructed for this example is

- Task $T_1(1, 3)$ with load 57 and $|57 - 52.5| \leq R$
- Task $T_2(4, 6)$ with load 48 and $|48 - 52.5| \leq R$
- Task $T_3(7, 10)$ with load 50 and $|50 - 52.5| \leq R$
- Task $T_4(11, 20)$ with load 55 and $|55 - 52.5| \leq R$

The accumulated load indeed is

$$57 + 48 + 50 + 55 = 210$$

5.6.2 PROOSA Service Control Implementation

Figure 5.5 shows a conceptual class diagram of the classes involved in the PROOSA system. The class PROOSAServiceControl uses an extended version of the “Cyberlink for C++” UPnP Control Point. This extended version called UPnPPROOSADevice derives from class ControlPoint provided Cyberlink. Class ControlPoint provides all functionality that is necessary for searching, discovering and using UPnP devices. However, since the PROOSA system is only interested in PROOSA devices and PROOSA services, class UPnPPROOSAControlPoint provides additional functionality with regard to PROOSA devices and services. Class UPnPPROOSAControlPoint provides an abstract layer that separates the UPnP part from the PROOSA part of the system.

Administration of the set of available PROOSA devices and services

Class ControlPoint is a UPnP control point and according to this protocol it adds each discovered device in the network to its set of available UPnP devices. Because we are only interested in PROOSA devices and services class UPnPPROOSAControlPoint keeps a set of discovered PROOSA devices that can be used. This is handled by overloading method *deviceSearchReponseReceived* of class ControlPoint. In this method the sub set of PROOSA devices is maintained. In case a PROOSA device leaves the network it is removed from the set of available PROOSA devices and services. This removal is performed in the overloaded method *onByeBye* of UPnPPROOSAControlPoint.

Event Handling

The completion results of actions of a PROOSA service are handled by class UPnPPROOSAControlPoint. It uses an EventListener to receive events from PROOSA services. Each time an event has been received from a service, method *eventNotifyReceived* is called. In this method the event is evaluated and the corresponding methods are called. The methods that are called, depending on the received event are

- *onInitialiseServicePerformed*
- *onSimulatePartialSRDExperimentPerformed*
- *onCombinePartialSRDOutcomePerformed*
- *onSimulateSFSExperimentPerformed*
- *onFinaliseServicePerformed*

which correspond with the action name that generated the event. Class PROOSAServiceControl inherits from UPnPPROOSAControlPoint and implements these methods.

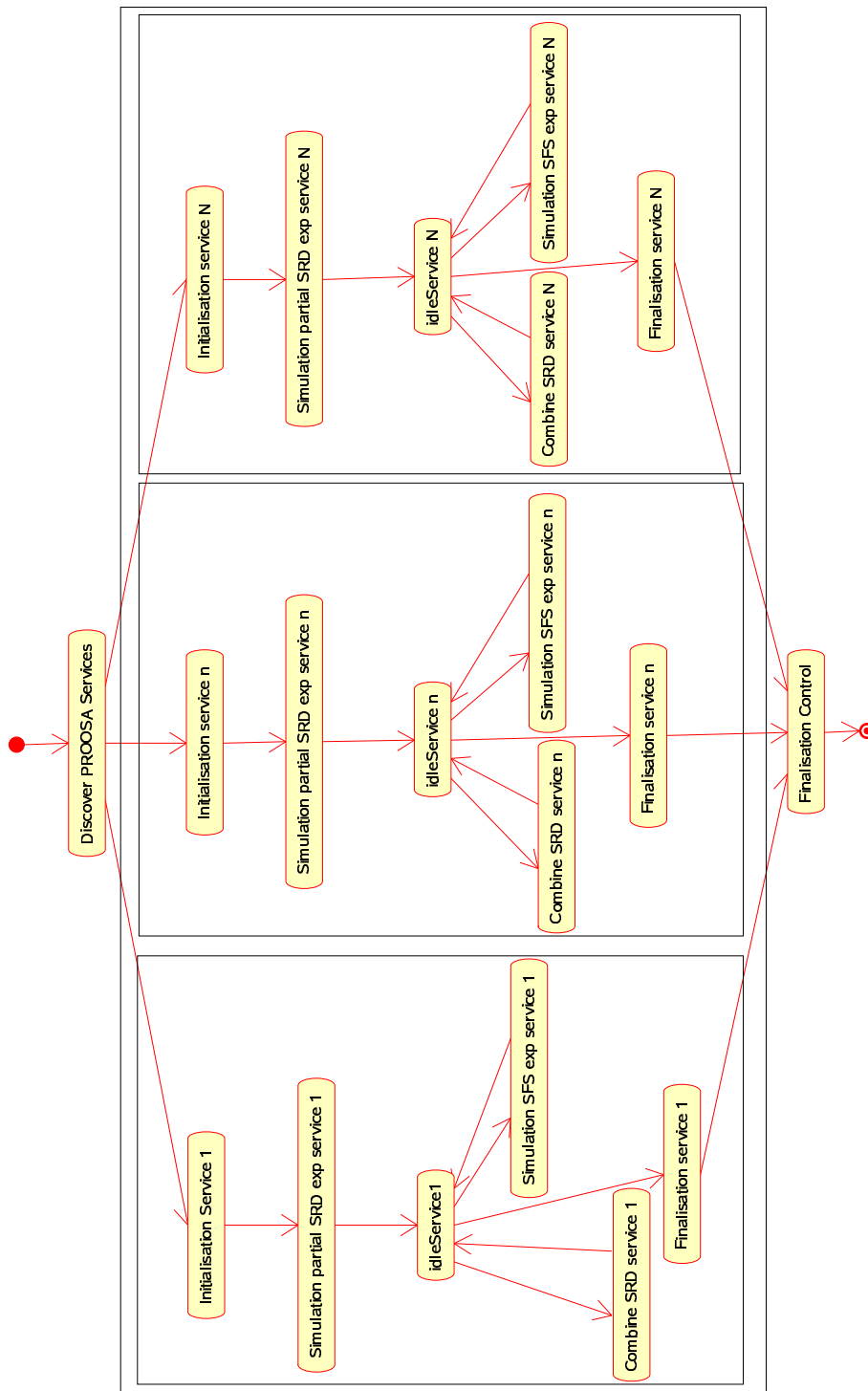


Figure 5.4: State Transition Diagram PROOSA service control

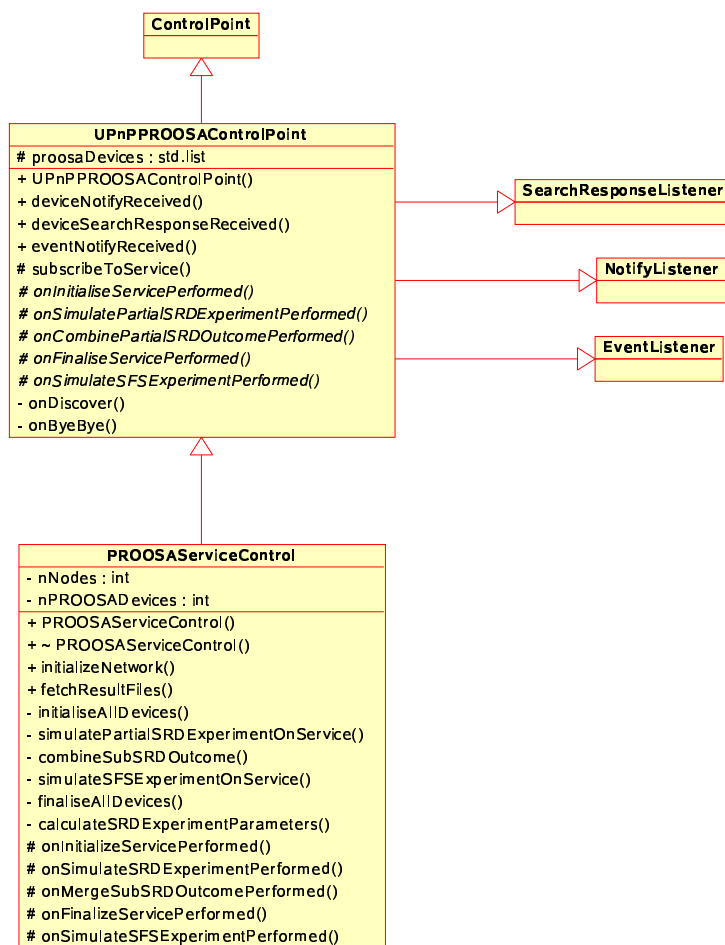


Figure 5.5: Conceptual class diagram of the PROOSA Service Control

Chapter 6

Performance Measurements

In this chapter we discuss the performance measurements for a number of scattering experiments. We measured total run times and execution times of particular phases both for the sequential version and for several parallel versions using distinct numbers of nodes. The test environment for these simulations is the SANDpit cluster at the Eindhoven University of Technology. This is a 17 node Beowulf cluster with an additional master node. The nodes are interconnected via Gigabit Ethernet.

First we will discuss the samples that are used for the simulations. Then the structure of the measurements is explained in detail. Finally an analysis of the measured values is given for an SFS experiment with an SRD experiment as prelude. Since simulations of MRD and SAXS/WAXS experiments are expected to show similar results they have not been performed.

6.1 Samples

The experiments that are performed use samples of NaCl which contain two substances, i.e. Na^+ ions and Cl^- ions. In the experiments, samples containing 8000, 16000, 32000, 64000 and 128000 particles are used. The particles of each sample are positioned in a box according to a Face Centred Cube (FCC) grid. The dimensions of these boxes are given in table 6.1

#particles in sample (10^3)	length (10^{-10})	width (10^{-10})	depth (10^{-10})
8	+56.28	+56.28	+56.28
16	+112.56	+56.28	+56.28
32	+112.56	+112.56	+56.28
64	+112.56	+112.56	+112.56
128	+225.12	+112.56	+112.56

Table 6.1: Box dimensions for samples used in simulations

6.2 Measurement structure

Recall from section 5.6 that for the execution of an SFS experiment with an SRD-prelude each service (node) executes a number of phases:

- initialisation tasks
- partial calculations of SRD outcomes tasks
- combination of partial SRD outcomes tasks
- simulation of SFS experiment task
- finalisation tasks

where each of these tasks is implemented as separate action of a service. In addition to the total execution time we are also interested in the duration of each of these actions. These time spans are given by $T_{initialisation}$ for the initialisation phase, $T_{partialSRD}$ for the partial SRD outcome calculation, $T_{combineSRD}$ for the combination of SRD outcomes, T_{SFS} for the calculation for an SFS outcome and $T_{finalisation}$ for the finalisation phase.

Since transition to a next phase does not happen at the same time for all services, we are interested in the time span between the start of the first action of a new phase and the completion of the last action of the previous phase. If this time span is negative it may, but need not, be the case that some service had been idle between two phases. Such a negative time span between phases is inevitable for strictly synchronised phases such as the combination and SFS calculation phases.

An example of both overlap and synchronisation of phases is illustrated in Figure 6.1 where three services are shown. Each service has a vertical life line on which the duration of each phase is shown. In this diagram there is no idle time between the

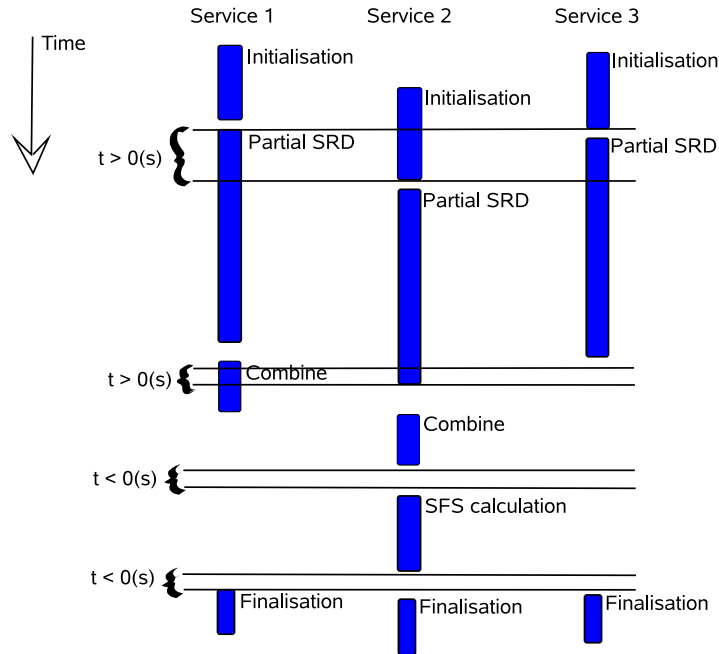


Figure 6.1: Overlap and synchronisation of phases

initialisation phase and SRD phase and between the SRD phase and Combine phase. Synchronisation occurs between the Combine phase and SFS phase and between the SFS phase and the finalisation phase.

This pattern is also visible in the numbers that are measured during the simulation runs. The time span between the initialisation and SRD phase and the time span

between the SRD phase and Combine phase is always positive. This means that there is no synchronisation between these phases as a whole. On the other hand, between phases that must be synchronised the numbers reflect this synchronisation by negative time spans.

6.3 Timing analysis

The execution times for runs of the sequential and parallel programs for several NaCl samples are measured using the Linux *time* command. This command runs the program and shows the real time, the user time and system time used by the program. Here we use the real time because this is the amount of time the user has to wait for the result. The execution time $T_{seq}(N)$ for the sequential program is dependent on the number of particles N in the sample. Based on the assumption that the SRD calculation, which is of $O(N^2)$ complexity, is the major computational part of the sequential program, we expect that execution times will increase with a factor 4 in case the workload is doubled. The numbers in Table 6.2 show that this is indeed the case.

The execution time $T_{par}(P, N)$ of the parallel program is dependent on the number of nodes P , i.e. services used for the calculation and the number of particles N in the sample. The execution times for the parallel program are listed in Table 6.2.

N	$T_{seq}(N)$	$T_{par}(1, N)$	$T_{par}(2, N)$	$T_{par}(4, N)$	$T_{par}(8, N)$	$T_{par}(16, N)$
8000	4.87	9.26	9.27	11.53	13.67	15.60
16000	17.10	21.42	18.41	19.48	24.85	28.73
32000	64.48	74.04	52.85	47.81	48.20	57.87
64000	257.10	302.503	193.41	128.89	114.00	119.67
128000	1017.51	1180.62	650.34	403.95	307.46	271.86

Table 6.2: Execution times (in seconds) of SRD + SFS simulations performed by the sequential and parallel program

The numbers for $T_{par}(1, N)$ follow roughly the same $O(N^2)$ pattern as the numbers for $T_{seq}(N)$. However, the numbers for $T_{par}(1, N)$ also contain initialisation time and the time needed for the transfer of the sample file, definition file, SRD outcome file and SFS outcome file. These accumulated times are roughly constant which is visible in the numbers.

In the simulations that are performed on 2 or more services, the two computationally expensive phases are the SRD phase and the Combine phase which is of $O(N)$ complexity. The execution times for the SRD phase are listed in Table 6.3 times and the execution times for the Combine phase are listed in Table 6.4. Except for $N = 8000$ the usage of more services for the SRD phase for a same problem delivers a speed up for the SRD phase. Since we implemented a recursive doubling strategy we expect the execution times of the Combine phase to growth logarithmically with the number of services used. This is indeed the case but note that the actual values are large.

The total execution time of a run of the parallel program is roughly the same as the accumulation of the execution time of the SRD phase and Combine phase for a run. For a small number of processes and a relatively small value for N this accumulated time is mainly dominated by the execution time of the SRD phase. On the other hand, if a large number of services is used the accumulated time is substantially influenced by the execution time of the Combine phase.

N	$SRD(2, N)$	$SRD(4, N)$	$SRD(8, N)$	$SRD(16, N)$
8000	3.57	2.72	2.81	3.20
16000	11.35	7.76	5.78	5.57
32000	39.76	23.79	15.81	12.76
64000	167.34	85.89	51.28	35.72
128000	601.37	316.46	181.03	107.29

Table 6.3: Measured times in seconds for SRD phases in the SRD + SFS experiments

N	$Com(2, N)$	$Com(4, N)$	$Com(8, N)$	$Com(16, N)$
8000	2.33	4.67	7.56	10.45
16000	4.63	9.65	14.56	19.68
32000	9.47	19.55	29.55	39.84
64000	19.33	39.06	58.90	79.53
128000	39.47	78.63	124.25	160.38

Table 6.4: Measured times in seconds for Combine phases in the SRD + SFS experiments

Using the performance numbers in Table 6.2, we are able to calculate some performance parameters known as the speed up and efficiency [18]. In this thesis the speed up is defined by

$$S(P, N) = \frac{T_{seq}(N)}{T_{par}(P, N)}$$

and the efficiency by

$$E(P, N) = \frac{S(P, N)}{P}$$

where

- N is the problem size, i.e. the number of particles in a sample
- $T_{seq}(N)$ is the sequential runtime on a sample with N particles
- $T_{par}(P)$ is the parallel runtime on same sample using P services (nodes)

Table 6.5 shows the numbers for the speed up based on numbers of Table 6.2.

N	$S(1, N)$	$S(2, N)$	$S(4, N)$	$S(8, N)$	$S(16, N)$
8000	0.53	0.52	0.42	0.36	0.31
16000	0.80	0.93	0.88	0.69	0.60
32000	0.87	1.22	1.35	1.34	1.11
64000	0.85	1.33	1.99	2.26	2.15
128000	0.86	1.56	2.52	3.31	3.74

Table 6.5: Speed up number based on Table 6.2

Table 6.5 shows that for $P \geq 2$ and $N \geq 32000$ it is useful to use a parallel program to perform an SFS experiment with an SRD experiment as prelude. If we look at the efficiency in Table 6.6 this table shows the efficiency is relatively poor as already indicated by Table 6.4. This is due to the large execution times of the Combine phase. Further research of the running times of the Combine phase revealed an interesting issue. It was expected, certainly for large sizes of an SRD file, that the time needed to transfer a partial outcome from one service to another would account for a major part of the execution time for the Combine phase. However, research of the log files of the FTP server showed that the transfer time was small, i.e. less than a second for SRD outcome files with a size of 40 mega bytes in experiments with 128000 particles. This revealed that another component in the Combine phase takes a lot of time. Analysis of the code revealed that a large amount of code using templates and casts is used for reading in values. It is expected that an optimisation is possible, thus enabling a faster combine phase and an improvement of the efficiency.

If, for instance for $N = 128000$ and $P = 16$, the execution time for the combine phase can be reduced from 160.38 seconds to roughly 80 seconds, the speed up would be equal to 5.3.

N	$E(1, N)$	$E(2, N)$	$E(4, N)$	$E(8, N)$	$E(16, N)$
8000	0.53	0.26	0.11	0.04	0.02
16000	0.80	0.46	0.22	0.09	0.04
32000	0.87	0.61	0.34	0.17	0.07
64000	0.85	0.66	0.50	0.28	0.13
128000	0.86	0.78	0.63	0.41	0.23

Table 6.6: Efficiency based on Table 6.5

Chapter 7

Conclusions

In this thesis we have shown that it is feasible to use a service oriented architecture for computationally intensive scientific applications. The advantage of this approach is that it makes it possible to harvest free resources available in a heterogeneous computing environment like, e.g. a set of PC's organised in a LAN.

The feasibility study involved three major parts. The first part was the identification of a suitable technology for a service oriented architecture. First we considered Grid technology but this was too heavy weight for our purposes. We also considered the JXTA technology but this technology has its main focus on the Java environment. We considered the installation effort for JXTA too large and our preliminary existing code was written in C++ so we wanted to have a solution that can be used with the C++ code directly. Therefore we chose to use the light weight, platform independent Universal Plug and Play protocol. We used the Cyberlink for C++ UPnP stack which could easily be integrated in existing code.

The second part of the feasibility study was the identification and adaptation of a computationally intensive application that could serve as a vehicle for the feasibility study. This vehicle was found in the domain of the simulation of scattering analysis experiments. As a starting point we used a preliminary object oriented model and sequential program for the simulation of scattering experiments. This adaptation consisted of the refactoring and design of an object oriented model and program. The refactoring of these elements consisted of the creation of a package structure and the removal over several Factory classes. Based on this sequential model a parallel model has been developed.

The last part of the feasibility study consisted of a set of tests to get a rough indication of the performance of the resulting system. First we ran tests on a Beowulf cluster and later we used 2 PC's in a local area network. Although speed has not been our major concern we have shown that in a homogeneous environment, i.e. a Beowulf cluster, modest speed ups can be realised. Initial analysis shows that a considerable improvement can be obtained if the costs of combination phase are lowered. Opportunities are in the improvement of code that uses casts to read in the partial SRD outcome file.

7.1 Future Work

It would be interesting to test the performance of the system for tasks of a finer grain size, such that the workload per task that calculates a partial SRD outcome is much smaller than the total workload divided by the number of services. Further it would be interesting to perform more detailed tests which can reveal bottlenecks more precisely

Another part of future research could be related to security of the system with regard to abuse of services and a secure file transfer protocol.

An introduction of a parser for definition files and parsers for outcomes of experiment would be an enhancement that yields a better design. These classes make it possible to decouple the I/O and object creation from the existing code.

The adaptation of code that combines partial histograms could deliver a considerable increase of the execution time of the combination.

Bibliography

- [1] http://en.wikipedia.org/wiki/upnp#api..2f_sdk.
- [2] http://sourceforge.net/project/showfiles.php?group_id=89768.
- [3] http://www.bipm.fr/en/si/base_units/.
- [4] <http://www.dftpd.com/products.php#indiftpd>.
- [5] http://www.gnu.org/software/libc/manual/html_node/posix-threads.html.
- [6] <http://www.ietf.org/internet-drafts/draft-ietf-secsh-filexfer-12.txt>.
- [7] <http://www.jxta.org>.
- [8] <http://www.trolltech.com/products/qt/index.html>.
- [9] http://www.upnp.org/download/upnpda10_20000613.htm.
- [10] <http://www.upnp.org/standardizeddcps/documents/dimmablelight1.0cc.pdf>.
- [11] <http://www.w3.org/protocols/rfc959/>.
- [12] Private communication initial oosa model with r. mak.
- [13] Tildesley D.J. Allen, M.P. *Computer Simulation of Liquids*. Oxford University Press, 1987.
- [14] R.J.J. Beckers. *Evaluation of UPnP in the context of Service Oriented Architectures*. Eindhoven University of Technology, Department of Mathematics and Computer Science, 2005.
- [15] Kesselman Foster and Tuecke. *The anatomy of the grid, enabling scalable virtual organizations*. 2001.
- [16] M. Fowler. *Analysis Patterns, Reusable Object Models*. Addison Wesley Longman, Inc, 1997.
- [17] Helm R. Johnson R. Vlissides J. Gamma, E. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1995.
- [18] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Publishing; 2nd edition, One Jacob Way, Reading, MA 01867-3999, second edition, January 2003.
- [19] Lichun Ji. *Computation in peer-to-peer networks*. 2004.
- [20] Sanders Beverly A. Massingil Berna L. Mattson, Timothy G. *Patterns For Parallel Programming*. Pearson Education, Inc., 2005.
- [21] R.P.T. van Gassel. *Parallel structure factor calculations*, 1995.
- [22] A.J.C. Wilson. *Intensity and interpretation of diffracted intensities, International Tables for Crystallography*. Kluwer Academic Publishers, 1989.

Appendix A

OOSA diagrams

A.1 OOSA Package Diagram

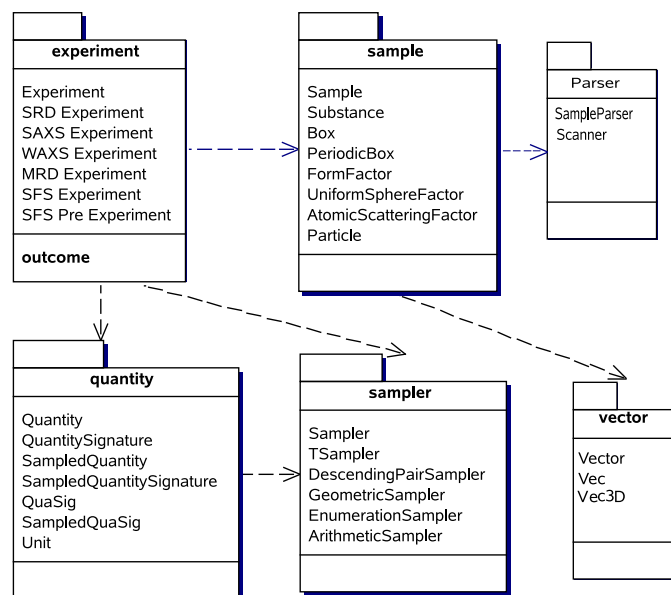


Figure A.1: OOSA package diagram