The Rubis asynchronous Web Service

Raymond Bisdorff

Computer Sciences and Communication Research Unit
Faculty of Sciences, Technology and Communication
University of Luxembourg

1 Introduction

This Manual (*Revision* : 1.4) describes the Rubis Web Service for computing best choice recommendations from bipolar-valued outranking digraphs. This computing resource is useful in the context of the testing of the Rubis decision support method [1].

Developing the Rubis decision support methodology is an ongoing research project of Raymond Bisdorff, University of Luxembourg.

The Rubis Web Service is based on the `digraphs` *Python* module (see http://ernst-schroeder.uni.lu/Digraph/) that requires at least version 2.4.0 of *Python*.

The basic idea of the Rubis Web Service is to distribute the Rubis progressive best choice decision method mainly to web services aware Decision-Deck clients (see the official Decision-Deck web site).

2 Purpose of this document

This document describes the architecture and implementation of the Rubis Python Service version 1[1]. Main components are the web service dispatcher, the jobs spool daemon and the Rubis MCDA solver.

The Rubis Python Server source code is copyrighted.

3 The Rubis client view

The Rubis web service follows the general recommendation of the Decision-Deck project (see Figure 3).

Three standard ports are recommended:

1. A `hello` port for testing the connection with the Rubis server machine.
2. A `submitProblem` port for submitting an XML encoded problem description.
3. A `requestSolution` port for requesting the XML encoded solution of the Rubis best choice decision method.

3.1 The `hello` port

The `hello` port is formally described in terms of WSDL-1.0. The port involves a question-response schema with a synchronous exchange of two messages of type string via a SOAP literal RPC encoding over HTTP.

[1]Forthcoming version 2 will be compliant with the XMCDA standards.

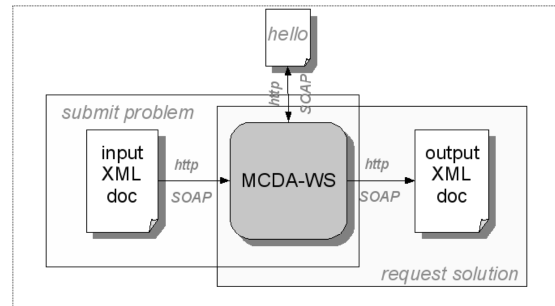

Figure 1: The Decision-Deck MCDA asynchronous web service layout

```
## http://ernst-schroeder.uni.lu/rubisServer/Version-1.2/rubisServer.wsdl
<description>

##  type declarations
  <types>
   <xsd:element name="hello">
     <xsd:complexType>
       <xsd:sequence>
       <xsd:element name="message"
                    minOccurs='0'
                    maxOccurs='1'
                    type="xsd:string" />
       </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="helloResponse">
    <xsd:complexType>
        <xsd:sequence>
    <xsd:element name="message"
                         minOccurs='0'
                         maxOccurs='1'
                         type="xsd:string" />
</xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </types>

##  message definitions
  <messages>
   <message name="hello">
     <part name="message" element="tns:hello"/>
   </message>

   <message name="helloResponse">
    <part name="message" element="tns:helloResponse"/>
   </message>
  </messages>
```

```
## port operation description
  <portType name="rubisServerPortType">
    <operation name="hello">
      <documentation>
       Returns the D3-Rubis server greetings.
      </documentation>
      <input message="tns:hello"
          name="hello" />
      <output message="tns:helloResponse"
           name="helloResponse" />
    </operation>
  </portType>

## port binding protocol description
  <binding name="rubisServerBinding" type="tns:rubisServerPortType">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="hello">
      <soap:operation soapAction="hello"/>
      <input name="hello">
      <soap:body use="literal" />
      </input>
      <output name="helloResponse">
  <soap:body use="literal" />
      </output>
    </operation>
  </binding>

## web service provider for the port
  <service name="rubisServer">
    <port name="rubisServerPortType" binding="tns:rubisServerBinding">
      <soap:address
         location="http://ernst-schroeder.uni.lu/cgi-bin/rubisServer.py"/>
    </port>
  </service>
</definitions>
```

Based on this WSDL description, the following *Python* code allows to verify for instance the connection with the ¡Ernst-Schroeder.uni.lu¿ RUBIS Server of the University of Luxembourg.

```
#!/usr/bin/env python
# using SOAPpy WSDL library for accessing
# the D3 Rubis Server from a WSDL description
# RB March 2008
#############################################

## import the generic web service tools
from SOAPpy import WSDL

## get a port proxy instance
url = 'http://ernst-schroeder.uni.lu/rubisServer/Version-1.2/rubisServer.wsdl'
rubisServer = WSDL.Proxy(url)
```

```
## call the remote hello method
message = rubisServer.hello()

## print hello response
print 'D3 Rubis server message:\n', message
```

The response from the RUBIS Server ¡ernst-schroeder.uni.lu¿ will be the following:

```
*********************************************
* This is the Ernst-Schroeder Apache Server *
* of the University of Luxembourg.          *
* Welcome to the Rubis Web services.        *
* RB March 2008, version 1.2                *
*********************************************
```

3.2 The `submitProblem` port

Similar to the hello port, we present hereafter the description of the `submitProblem` port. The same question-response type of message exchange is again implemented. The question consists of the XML encoded `problemFile`. The RUBIS server responds with a message acknowledging the good reception of the problem file and a ticket allowing later on to request the corresponding XML encoded solution file. We use again an RPC literal encoded SOAP exchange over standard HTTP.

```
##  type declarations
<types>
    <xsd:element name="submitProblem">
<xsd:complexType>
            <xsd:sequence>
    <xsd:element name="problemFile"
                         minOccurs='0'
                         maxOccurs='1'
                         type="xsd:string" />
   </xsd:sequence>
</xsd:complexType>
     </xsd:element>
     <xsd:element name="submitProblemResponse">
<xsd:complexType>
           <xsd:sequence>
    <xsd:element name="message"
                         minOccurs='0'
                         maxOccurs='1'
                         type="xsd:string" />
    <xsd:element name="ticket"
                         minOccurs='0'
                         maxOccurs='1'
                         type="xsd:string" />
   </xsd:sequence>
</xsd:complexType>
     </xsd:element>
</types>

##  message definitions
```

```
<messages>
  <message name="submitProblem">
   <part name="problemFile"
         element="tns:submitProblem"/>
  </message>
  <message name="submitProblemResponse">
    <part name="message"
          element="tns:submitProblemResponse"/>
    <part name="ticket"
          element="tns:submitProblemResponse"/>
  </message>
</messages>

## port operation description
<portType name="rubisServerPortType">
    <operation name="submitProblem">
      <documentation>
       Submit a Rubis performance tableau version 0.1. Get a ticket in return.
      </documentation>
      <input message="tns:submitProblem"
             name="submitProblem" />
      <output message="tns:submitProblemResponse"
               name="submitProblemResponse" />
    </operation>

   </portType>

## port binding protocol description
<binding name="rubisServerBinding" type="tns:rubisServerPortType">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
   <binding name="rubisServerBinding" type="tns:rubisServerPortType">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="submitProblem">
      <soap:operation soapAction="submitProblem"/>
      <input name="submitProblem">
      <soap:body use="literal" />
      </input>
      <output name="submitProblemResponse">
  <soap:body use="literal" />
      </output>
    </operation>
<binding>
```

An example of XML encoded problem file may be consulted at the following url: "http://ernst-schroeder.uni.lu/rubisServerDoc/problemDefinition.xml" server.

The `submitProblem` port may be accessed with the following *Python* client code:

```
#!/usr/bin/env python
# using SOAPpy WSDL library for accessing
# the Rubis Server port submitProblem
# RB March 2008
#########################################
```

```
# import the generated class stubs
from SOAPpy import WSDL
import SOAPpy
# get a port proxy instance
url = 'http://ernst-schroeder.uni.lu/rubisServer/Version-1.1/rubisServer-1.1-0.wsdl'
rubisServer = WSDL.Proxy(url)

## get xmlProblemFile
fileIN = open('problemDefinition.xml','r')
problemText = fileIN.read()
fileIN.close()

## call the remote submitProblem method
answer = rubisServer.submitRubisProblem(problemFile=problemText)

## print request results
print 'Rubis server message:\n', answer['message']
print 'Rubis server ticket:', answer['ticket']
```

The successful response of the server is composed of two items: – the server message, and – the ticket identifying the submitted job. An example of response, when running the *Python* code above is given below:

```
Rubis server message:
The problem submission was successful !
Rubis server ticket: quJlfsRBr8ohgbtL
```

The server ticket `quJlfsRBr8ohgbtL` is required for the subsequent solution request.

3.3 The `requestSolution` port

Similarly to both the ports already described we give here the WSDL-1.0 definition of the standard `requestSolution` port.

```
##  type declarations
<types>
   <xsd:element name="requestSolution">
      <xsd:complexType>
         <xsd:sequence>
    <xsd:element name="ticket"
                         minOccurs='0'
                         maxOccurs='1'
                         type="xsd:string" />
   </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="requestSolutionResponse">
      <xsd:complexType>
         <xsd:sequence>
    <xsd:element name="ticket"
                         minOccurs='0'
```

```
                        maxOccurs='1'
                        type="xsd:id" />
    <xsd:element name="message"
                        minOccurs='0'
                        maxOccurs='1'
                        type="xsd:string" />
    <xsd:element name="solution"
                        minOccurs='0'
                        maxOccurs='1'
                        type="xsd:string" />
 </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</types>

##  message definitions
<messages>
  <message name="requestSolution">
   <part name="ticket"
         element="tns:requestSolution"/>
  </message>
  <message name="requestSolutionResponse">
    <part name="ticket"
          element="tns:requestSolutionResponse"/>
    <part name="message"
          element="tns:requestSolutionResponse"/>
    <part name="solution"
          element="tns:requestSolutionResponse"/>
  </message>
</messages>

## port operation description
<portType name="rubisServerPortType">
    <operation name="requestSolution">
      <documentation>
       Request a Rubis outranking digraph with a given ticket.
      </documentation>
      <input message="tns:requestSolution"
             name="requestSolution" />
      <output message="tns:requestSolutionResponse"
              name="requestSolutionResponse" />
    </operation>
</portType>

## port binding protocol description
<binding name="rubisServerBinding" type="tns:rubisServerPortType">
    <operation name="requestSolution">
      <soap:operation soapAction="requestSolution"/>
      <input name="requestSolution">
      <soap:body use="literal" />
      </input>
      <output name="requestSolutionResponse">
```

```
  <soap:body use="literal" />
      </output>
    </operation>
</binding>
```

The XML encoded solution of the RUBIS solver may be accessed with the following *Python* client code, where we reuse the server ticket returned at the submission of the problem.

```
#!/usr/bin/env python
# using SOAPpy WSDL library for accessing
# the Rubis Server
# RB March 2008
#########################################

# import the generated class stubs
from SOAPpy import WSDL
import SOAPpy
# get a port proxy instance
url = 'http://ernst-schroeder/rubisServer/Version-1.1/rubisServer-1.1-0.wsdl'
rubisServer = WSDL.Proxy(url)

## t1 = answer['ticket'] from a previous submitProblem access
t1 = 'quJlfsRBr8ohgbtL'
answer = rubisServer.requestOutrankingDigraph(ticket=t1)

# print request results
print 'Rubis server message:\n', answer['message']
if answer['message'] != 'Solution not available !!':
    problemText = answer['solution']
    fileName = 'problemSolution.xml'
    fileOUT = open(fileName,'w')
    fileOUT.write(problemText)
    fileOUT.close()
    print 'See file: ', fileName
```

The XML encoded solution file may be consulted at the following url: "http://ernst-schroeder.uni.lu/rubisServerDoc/problemSolution.xml" server.

4 The Rubis Server layout

In this section we describe the RUBIS server layout as it is actually implemented on the *ernst-schroeder.uni.lu* server at the University of Luxembourg.

The general layout of the RUBIS Server (see Figure 2) uses the following directories that are supposed to be placed in the `/var`[2] directory under the RUBIS Server root directory: `/var/rubisServerData/`.

Main components of the RUBIS server are: – the web service ports dispatcher written in *Python* and based on the `ZSI` module, – the job spool daemon written in `bash`, and – the RUBIS problem solver again written in *Python* and based on the `digraphs` module.

[2]The ¡ernst-schroeder.uni.lu¿ server is running under Ubuntu 7.10 Gutsy following a Debian Linux recommended general directory layout. The Apache server's public pages are located in the `/var/www` directory.

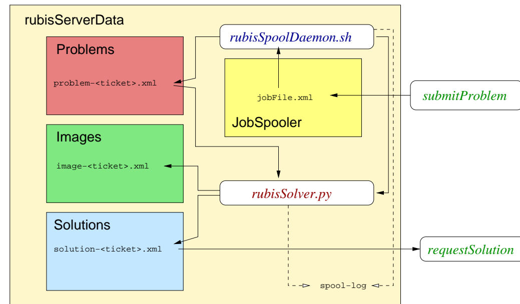

Figure 2: The Rubis Server layout

4.1 The service ports dispatcher

The Rubis web service ports are dispatched with the help of the ZSI *Python* module. The following *Python* program is located in the `cgi-bin` directory of the Apache server on ¡enrst-schreoder.uni.lu¿ and can be accessed via the http://ernst-schroeder.uni.lu/cgi-bin/rubisServer-1.1-0.py URL.

The three Decision-Deck standard service ports, namely `hello`, `submitProblem`, and `requestSolution`, are implemented as *Python* functions returning a structured answer containing the required messages as mentioned in Section 3.

```
#!/usr/bin/python
# rubisServer.py
# RB March 2008
###########################

## hello port implementation
1: def hello():
2:     return {"message":
3:     """
4:     *********************************************
5:     * This is the Ernst-Schroeder Apache Server *
6:     * of the University of Luxembourg.          *
7:     * Welcome to the Rubis Web services.        *
8:     * RB March 2008, version 1.1-0              *
9:     *********************************************
10:    """}

## submitProblem port implementation
11: def submitProblem(**kw):
12:     import random, string, codecs
13:     try: # try saving of submitted problem file
14:      chars = string.letters+string.digits
15:         ticket = ''.join([random.choice(chars) for i in range(16)])
16:         fileName = '/var/rubisServerData/JobSpooler/problem-'+ticket+'.xml'
17:         fileOUT = codecs.open(fileName,'w',encoding='utf-8')
18:         fileOUT.write(unicode(kw['problemFile']))
19:         fileOUT.close()
20:         msg = 'The problem submission was successful !\n'
21:         return {"message": msg, "ticket": ticket}
22:     except:
```

```
23:         return {"message": "The problem submisson was unsuccessful",
                "ticket":None}

## requestSolution port impementation
24: def requestSolution(**kw):
25:     ticket = kw['ticket']
26:     answer = {}
27:     try: # check availability of solution file
28:         fileName = '/var/rubisServerData/Solutions/solution-'\
29:                     +str(ticket)+'.xml'
30:         fileIN = open(fileName,'r')
31:         answer['solution'] = fileIN.read()
32:         fileIN.close()
33:         msg = "The solution request was successful.\n"
34:     except:
35:         msg =  "Solution not available !!"
36:     answer['ticket'] = ticket
37:     answer['message'] = msg
39:     return answer

## CGI Python script
40: if __name__ == '__main__':
41:     import os, sys
42:     from ZSI import dispatch
43:     dispatch.AsCGI(rpc=True)
```

Lines 1–10 implement the `hello` server response.

Lines 11–23 implement the `submitProblem` port operation. In Line 15 first a 16 character long server ticket is generated and the XML encoded submitted problem file is written to the Problems directory with name `problem-<ticket>.xml` (see Figure 2). The answer containing a message and the actual response ticket is returned as port response.

Lines 24–39 implement the `requestSolution` port operation. If the solution is available in the `Solutions` directory, it is returned as port response, otherwise a message "*Solution not available!!*" is returned.

Lines 40–43 implement the corresponding SOAP rpc literal service as a CGI *Python* script using the *Python* `ZSI` module.

4.2 The Rubis spool daemon

To liberate the `submitProblem` port from the eventual finishing of the Rubis problem solver, the submitted problem files are written to the `JobSpooler` directory (see Figure 2) which is watched for incoming jobs by the Rubis spool daemon, a `bash` script shown below.

```
#!/bin/bash
## rubisSpoolDaemon.sh
## RB March 2008
## version 1.1
####################

# Rubis Server installation directory on Debian/Linux Ubuntu 7.10 Gutsy
1: rootdir="/var/rubisServerData"
```

```
#  clean up when stopping the spool server
2: function cleanup {
3:     echo $(date) ' Stopping the Spool Server!' >> $rootdir/spool-log
4:     date >> $rootdir/spool-log
5:     kill "$@" $(jobs -p)
6:     exit
7: }

8: trap cleanup INT TERM
9: trap "kill -s TERM $(jobs -p); exit" INT KILL


#  log the spool daemon starting command
11: echo $(date) ':  Rubis spool daemon started.' >> $rootdir/spool-log

#  main spooling loop
12: while true
13: do
14:       for jobFile in $rootdir/JobSpooler/*; do
15:           if [ -f $jobFile ]; then
16:              mv $jobFile $rootdir/Problems
                 ## slice the ticket out of jobFile
17:           ticket=${jobFile:40:16}
                 ## standard Rubis Solver
18:              python $rootdir/executeRubisSolver.py $ticket &
19:              echo $(date) ': ' $jobFile 'spooled.' >> $rootdir/spool-log
20:           fi
21:           sleep 1
22:       done
23: done
```

Line 1 specifies the Rubis Server installation directory. Lines 2 – 9 define a clean up function to be run when stopping the spool daemon. Upon getting the TERM signal the spool daemon will as well send a TERM signal to all its still running child processes.

The main spool loop, in Lines 12 – 23, is running for ever. For each detected entry in the `JobSpooler` directory (Line 14), if it is a file (Line 15) it will be first moved to the `Problems` directory (Line 16). Then, in Line 17, the ticket of the job is sliced out and in Line 18, the Rubis solver, with the previous ticket as argument, is started in the background. Finally, a log entry is written to the standard `spool-log` file.

The spool daemon is started like a `SysV` service on booting the server as shown in the `bash` script below.

```
#!/bin/bash
## rubis
## RB March
#############

## switching on first argument $1
1: if [ -z $1 ]; then
2:  echo 'Usage: rubis  start | stop'
3: elif [ $1 = start ]; then
4:         pid=$(ps ax | grep rubisSpoolDaemon | grep sh)
```

```
5:         if [ -z $pid ]; then
6:  /var/rubisServerData/rubisSpoolDaemon.sh &
7:          pid=$(ps ax | grep rubisSpoolDaemon | grep bash)
8:          echo "Rubis Spool Daemon started now with pid: ${pid%%\ *}"
9:  else
10: echo "Rubis Spool Daemon already started with pid: ${pid%%\ *}"
11:  fi
12: elif [ $1 = stop ]; then
13:         ## find rubisSpoolDaemon pid
14:         pidline=$(ps ax | grep rubisSpoolDaemon | grep sh)
15:         echo $pidline | { read pid rest ;\
16:                           echo  "Rubis Spool Daemon with pid = $pid";\
17:                           echo "stopping now !";\
18:                           kill $pid; \
19:                        }
20: else
21:      echo 'Usage: rubis  start | stop'
22: fi
```

The RUBIS spool daemon may thus be started (Lines 3 – 11) or stopped (Lines 12 – 19) with the classic command `$/etc/init.d/rubis start | stop`.

4.3 The Rubis problem solver

The RUBIS solver is implemented in *Python* and is based upon the `digraphs` *Python* module (see http://ernst-schroeder.uni.lu/Digraph). The script below illustrates the general commands required for solving a best choice decision problem.

```
#!/usr/bin/env python
# rubisSolver.py
# RB March 2008
#################

1: import digraphs

2: try:
3:  ticket = sys.argv[1]
4: problemFileName = '/var/rubisServerData/Problems/problem-'\
                          + str(ticket)
5: t = digraphs.XMLRubisPerformanceTableau(problemFileName)
6: g = digraphs.BipolarOutrankingDigraph(t)
7: solutionFileName = '/var/rubisServerData/Solutions/outrankingDigraph-'\
                           + str(ticket)
8: g.saveXMLRubisOutrankingDigraph(name=solutionFileName,\
                                       category=t.category,\
                                       subcategory=t.subcategory,\
                                       author=t.author,\
                                       reference=t.reference,\
                                       noSilent=False)
9: graphFileName = '/var/rubisServerData/Images/rubisGraph-' + str(ticket)
10: g.exportGraphViz(fileName=graphFileName,\
                        graphSize='10,10',\
                        graphType='png',\
```

```
                          bestChoice=g.goodChoices[0][5],\
                          worstChoice=g.badChoices[0][5])
11: except:
12: solutionFileName = '/var/rubisServerData/Solutions/outrankingDigraph-'
                             + str(ticket) + '.xml'
13: OUT = open(solutionFileName,'w')
14: OUT.write('Error: rubis Solver could not compute\
                    the requested outranking digraph!')
15: OUT.close()
```

In Line 1, the `digraphs` module is imported. We first try to solve the submitted problem (Lines 2 – 10). If this fails for some reason, an error message is returned instead of the solution file (see Lines 11 – 15).

Solving the problem is actually achieved through Lines 3 – 10, where the problem file name is constructed with the help of the ticket input parameter (Lines 3 – 4). The XML encoded Rubis performance tableau is then read into the *Python* variable `t` (Line 5) from the `Problems` directory.

In Line 6 we construct the bipolar-valued outranking digraph `g` from the given performance tableau `t`. After constructing the solution file name, we save the XML encoded complete Rubis results into the `Solutions` directory (Lines 7 – 8).

Finally, the corresponding graph file name is constructed (Line 9) and a `GraphViz` generated image in `PNG` format is saved in the `Images` directory. The most determined best and worst choice recommendations are marked golden respectively blue in the resulting image (Line 10).

For a precise documentation of the Rubis solver, it is helpful to directly consult the source code of these methods in the `digraphs` module source code. More information is available on the `digraphs` documentation pages (see http://ernst-schroeder.uni.lu/Digraph/).

5 Version comments

Features to come

- XMCDA compliant input and output schemes are foreseen for version 2.0.
- Electre 3 Rubis solver

Release 1.2 Actual features:

- Standard Decision-Deck compliant MCDA Web service
- Rubis jobs spool daemon and sysv service command
- Rubis solver *Python* source code

6 Acknowledgments

Thanks to everybody who reported bugs or suggested useful new features, but especially to Gilles Godinet from Karmic Sotware Research (Paris).

7 Copyright

The *Python*Rubis Web Service code source is “free,” this means that everyone is free to use it and free to redistribute it on certain conditions. The code is not in the public domain; it is copyrighted and there are restrictions on its distribution as follows:

Copyright © 2008 Raymond Bisdorff (University of Luxembourg)

All source code shown here is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This resource is distributed in the hope that it will be useful, but *without any warranty*; without even the implied warranty of *merchantability* or *fitness for a particular purpose*. See the GNU General Public License for more details. A copy of the GNU General Public License is available on the World Wide web.[3] You can also obtain it by writing to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

References

[1] Raymond Bisdorff, Marc Pirlot and Marc Roubens, *On Choices and kernels in bipolar valued digraphs. European Journal of Operational Research (EJOR)*, 175 (2006) 155-170.

[2] Raymond Bisdorff,Patrick Meyer, Marc Roubens, *RuBy: a bipolar valued outranking methodology for the best choice decision problem*, SMA Preprints 02 version 01, University of Luxembourg (2006), *PDF*.

[3] Béla Bollobás, *Random Graphs* (2nd edition). Cambridge University Press, 2001.

[3]at `http://www.gnu.org/copyleft/gpl.html`

Index

Figures, 1
- Asynchronous service layout, 2
- Server layout, 9

Ports dispatcher, 9
problem file, 5

requestSolution port
- solution file, 8

Rubis solver, 12

server ticket, 6, 8
Spool daemon, 10
sysv rubis service, 11

Web service
- description, 1
- hello port, 1
- purpose, 1
- requestSolution port, 6
- server design, 8
- submitProblem port, 4

Contents

1 Introduction **1**

2 Purpose of this document **1**

3 The Rubis client view **1**
3.1 The `hello` port 1
3.2 The `submitProblem` port 4
3.3 The `requestSolution` port 6

4 The Rubis Server layout **8**
4.1 The service ports dispatcher 9
4.2 The RUBIS spool daemon 10
4.3 The RUBIS problem solver 12

5 Version comments **13**

6 Acknowledgments **13**

7 Copyright **13**