RUBLX : A RUBY-BASED BATCH LANGUAGE FOR XGRID

Tetsuya SUZUKI
Kiyoto HAMANO
Department of Electronic Information Systems
Shibaura Institute of Technology
Minuma, Saitama city, Saitama,
337-8570, Japan
Email: {tetsuya, m107076}@sic.shibaura-it.ac.jp

Abstract— **We present a Ruby-based batch language for Xgrid and its processor. Xgrid is an environment for distributed and parallel computing on the Mac OS X operating system, and Ruby is an object-oriented programming language for general purposes. In the standard Xgrid environment, jobs in batch files are statically defined by an XML-based language, and submitted jobs are managed by their ID numbers. It is not easy for human to read and write XML-based batch files and to manage jobs by ID numbers. In our approach, jobs in batch files can be dynamically defined by a Ruby-based language, and submitted jobs can be managed by their logical names. Semantic checks and consistency managements are also done at submission in our approach. Our approach syntactically and semantically makes it easy to use Xgrid.**

Keywords— **Grid and Cluster Computing, Languages**

I. Introduction

Xgrid[1] is an environment for distributed and parallel computing on the Mac OS X operating system. We use Xgrid for Web mining and metadata generation[5], which need much computation. By the standard method, jobs for Xgrid can be submitted by an XML-based batch language and are managed by their job ID numbers.

We have difficulties in reading and writing the XML-based batch files and managing jobs. XML tags in batch files and job management by IDs are obstacles for them.

To solve the problems, we present a Ruby-based batch language for Xgrid and its processor. Ruby[2] is an object-oriented programming language. By our method, jobs can be submitted by concise batch files and be managed by symbolic names.

The organization of this paper is as follows. In section 2 we explain Xgrid and its batch language. We present our approach using examples in section 3, and compare it with other approaches in section 4. In section 5 we state our conclusion. We explain the detail of our batch language in appendix.

II. Xgrid and the batch language

A. Xgrid

Xgrid mainly consists of three kinds of software: clients, a controller and agents. A *client* is a program to submit jobs to a controller and receive the results from it. A *job* is a set of tasks, and a *task* is a program executed in Xgrid. A *controller* is a program to receive jobs from clients, split them into tasks and send them to agents. An *agent* is a program to execute assigned tasks and return the results to the controller. The results are sent to clients via the controller. They can work on different computers connected by local area network.

```
% xgrid -h xgridcontroller -p pass
        -job batch bc.plist
```

Fig. 1. Job submission by 'xgrid'

```
% xgrid -h xgridcontroller -p pass
        -job results -id 381
```

Fig. 2. Retrieval of the results by 'xgrid'

The standard client program 'xgrid', which is invoked from command line interface, provides two methods to submit jobs. In the first method we specify a job, which consists of a program and its command line arguments, in the command line arguments for 'xgrid'. We can submit a single task job only at once by the method. In the second method we specify a batch file in the command line arguments for 'xgrid'. A *batch file* is a file to specify jobs and their tasks. In batch files we can also describe jobs which must finish before a job starts, and tasks which must finish before a task starts. We call such relations *dependency relationships*. We can submit multi-task jobs at once by the method. The client program 'xgrid' also provides methods to manage jobs and retrieve their results using job ID numbers. For example, we can stop, delete and restart jobs.

Fig.1 and Fig.2 show how to use the 'xgrid' command from command line. Fig.1 is an example of job submission where the argument 'bc.plist' is a batch file name. Fig.2 is an example of retrieval of the results where the argument '381' is the job ID whose results is retrieved. In both cases the '-h' and the '-p' options specify a controller's host name and a password to connect it respectively.

B. The standard batch language

The standard batch language for Xgrid is based on XML with key/value structure. We explain the language using Fig.3 in the following.

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

Fig.3 shows a batch file in the batch language. It defines a job with a task which executes a command line '/usr/bin/bc -q bc_exp.txt' in Xgrid where a calculator program '/usr/bin/bc' reads an expression '1+2' from the script file 'bc_exp.txt' of Fig.4 and outputs the value of the expression to its standard output. The script file 'bc_exp.txt' is defined from the line 8 to the line 18 of Fig.3. The contents of the script file is embedded in the line 13 of Fig.3 as a base64-encoded string. Base64[3] is an encoding method which translates a byte stream to a US-ASCII string. The task is defined from the line 19 to the line 31 of Fig.3. The command path and its command line arguments are defined there.

As shown in Fig.3, users are responsible for consistency managements such that a job definition must include all file definitions which tasks in the job will refer to.

The batch language provides task prototype for concise task definitions though it is not used in Fig.3.

C. Problems in use of Xgrid

The followings are problems in use of Xgrid.

1. XML tags are obstacles for human to read and write batch files.
2. Files referred by tasks must be embedded as base64-encoded strings in batch files.
3. Jobs can not be dynamically determined by batch files at submission. In cases such that a job to execute many of a program with different parameters is described, the batch file can be more concise by dynamically determined jobs.
4. Jobs are managed by their ID numbers.
5. Dependency relationships among jobs in a same batch file can not be specified. The relationships must be specified by job IDs determined at submission which are never known at describing the batch file.
6. Xgrid users are responsible for semantic consistency check of batch files.

III. A Ruby-based batch language and its processor

To solve the problems pointed out in the previous section, we propose a Ruby-based batch language for Xgrid (RuBLX) and a client program 'rxgrid' for RuBLX. In this section, we explain the design principles and examples of batch files in RuBLX. We explain the detail of our batch language in appendix.

A. Design principles

The followings are design principles for our approach.

1. XML tags are not used in batch files.
2. Base64 encoding are not needed in batch files.
3. Jobs can be dynamically determined.
4. Jobs can be managed by symbolic names.
5. Dependency relationships among jobs in a same batch file can be specified.

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <!DOCTYPE plist PUBLIC
"-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
 3: <plist version="1.0">
 4:  <array>
 5:   <dict>
 6:    <key>name</key>
 7:    <string>job1</string>
 8:    <key>inputFiles</key>
 9:    <dict>
10:     <key>bc_exp.txt</key>
11:     <dict>
12:      <key>fileData</key>
13:      <data>MSArIDIKcXVpdAo=
14:      </data>
15:      <key>isExecutable</key>
16:      <string>NO</string>
17:     </dict>
18:    </dict>
19:    <key>taskSpecifications</key>
20:    <dict>
21:      <key>bc</key>
22:      <dict>
23:       <key>command</key>
24:       <string>/usr/bin/bc</string>
25:       <key>arguments</key>
26:       <array>
27:        <string>-q</string>
28:        <string>bc_exp.txt</string>
29:       </array>
30:      </dict>
31:    </dict>
32:   </dict>
33:  </array>
34: </plist>
```

Fig. 3. An XML-based batch file 'bc.plist'

```
1 + 2
quit
```

Fig. 4. A script file 'bc_exp.txt'

6. Semantic checks and consistency management are automatically done.

We use the Ruby programming language as the basis of our batch language because Ruby can evaluate a string as its program. Our client program 'rxgrid' implemented in Ruby evaluates an RuBLX batch file as a Ruby program for lexical analysis, syntax analysis and semantic analysis. It then generates XML-based batch files where base64 encoding and semantic check are automatically done, and submits them using 'xgrid'. The client generates an XML-based batch file for each job in a same RuBLX batch file to enable to specify dependency relationships among jobs in the RuBLX batch file. It also generates a map file, which records the correspondence between job IDs and symbolic job names, at submission. The client provides job management methods by symbolic job names using map files.

B. Examples

We explain an overview of our approach and show that our approach solves the problems pointed out in the previous section using three examples.

```
 1: file "bc_exp.txt" do |t|
 2:   t.agentPathName = "bc_exp.txt"
 3:   t.localPathName = "bc_exp.txt"
 4:   t.isExecutable = false
 5: end
 6:
 7: task "bc" do |t|
 8:   t.command = "/usr/bin/bc"
 9:   t.arguments = ["-q", "bc_exp.txt"]
10:   t.refersTo = ["bc_exp.txt"]
11: end
12:
13: job "job1" do |t|
14:   t.tasks = ["bc"]
15: end
```

Fig. 5. An RuBLX batch file 'bc.rb'

```
381,job1
```

Fig. 6. A map file 'bc_map.csv'

```
% rxgrid -h xgridcontroller -p pass
         -job batch bc.rb
```

Fig. 7. Job submission

```
% rxgrid -h xgridcontroller -p pass
         -job results -id job1
         -map bc_map.csv
```

Fig. 8. Retrieval of the results

The first example is shown in Fig.5. It is an example of a batch file in RuBLX. The job described there is the same as the job of Fig.3. Files, tasks and jobs have logical names as identifiers, and are defined as follows.

- A file is defined from the line 1 to the line 5. It starts with a keyword 'file' followed by a logical file name 'bc_exp.txt' and a do-end block with a parameter 't'. The followings are defined in the block: a path name of the file on agent machines, the contents of the file by a local file and whether it is executable or not.
- A task is defined from the line 7 to the line 11. It starts with a keyword 'task' followed by a logical task name 'bc' and a do-end block with a parameter 't'. The followings are defined in the block: a path name of a command, command line arguments for it and a file referred by it,
- A job is defined from the line 13 to the line 15. It starts with a keyword 'job' followed by a logical job name 'job1' and a do-end block with a parameter 't'. A task in the job is defined in the block.

The batch file in Fig.5 is more concise than the batch file in Fig.3. There is no XML tag in Fig.5, and the number of lines in Fig.5 is 15 while that in Fig.3 is 34.

A consistent management is done at submission of the batch file. The job definition in the generated XML-based batch file includes a file definition which the task 'bc' refers to though the file definition is not referred in the job definition in the RuBLX batch file.

Fig.6 shows a map file generated at submission of the batch file of Fig.5. The map file indicates that the ID number of the job 'job1' is 381.

Fig.7 and Fig.8, which correspond to Fig.1 and Fig.2 respectively, show how to use our processor 'rxgrid'. Fig.7 shows an example of job submission where the command line argument 'bc.rb' is a batch file name. Fig.8 shows an example of retrieval of the results where the command line arguments 'job1' and 'bc_map.csv' are the job name whose results is retrieved and a map file respectively.

The second example is shown in Fig.9. It is an example of a batch file with dependency relationships among jobs. Three jobs 'job0', 'job1' and 'job2' are defined there.

- The job 'job0' is a previously submitted job with the job ID 333. The job ID can be specified by a pair of a logical job name and a map file as shown in appendix.
- The job 'job1' is a job with a task 'echo1'.
- The job 'job2' is a job with a task 'echo2' which starts after two jobs 'job0' and 'job1' are done. The dependency is defined by 't.dependsOnJobs' in the block.

Our client program takes account of both dependency relationships among jobs and those among tasks. It does topological sort on jobs for submission. If it finds either cyclic dependency relationships for jobs or those for tasks, it declares errors.

The third example is shown in Fig.10. It is an example of a batch file where the number of tasks in a job are dynamically determined at submission.

In the batch file, a task is defined for each file whose name ends with '.txt' in a current directory. The files are collected by a standard Ruby library 'Dir' in the line 1 of Fig.10. Each task calculates the value of an expression in the file using '/usr/bin/bc'. Variables, arrays, flow controls and a standard Ruby library are used in the batch file because it is not known how many files will exist in a current directory at submission when the batch file is written.

Definitions of files, tasks and jobs are basically declarative in RuBLX. The order of definitions is not significant. Procedural description, however, can be used as this example.

Templates can be used for definitions in RuBLX though some programming skill is needed. For example, a template for a task is used from the line 16 to the line 20 in Fig.10 where 'taskName' and 'f.to_s' are used as parameters for the template.

IV. Comparison

We compare our approach with PyXG[1] because our approach solves problems of the standard Xgrid

```
 1: task "echo1" do |t|
 2:   t.command = "/bin/echo"
 3:   t.arguments = ["1"]
 4: end
 5:
 6: task "echo2" do |t|
 7:   t.command = "/bin/echo"
 8:   t.arguments = ["2"]
 9: end
10:
11: job "job0" do |t|
12:   t.id = 333
13: end
14:
15: job "job1" do |t|
16:   t.tasks = ["echo1"]
17: end
18:
19: job "job2" do |t|
20:   t.tasks = ["echo2"]
21:   t.dependsOnJobs = ["job0", "job1"]
22: end
```

Fig. 9. An RuBLX batch file with dependency relationships among jobs

environment which we pointed out as shown in the previous section, and PyXG also uses a programming language to specify and submit Xgrid jobs as our approach.

PyXG is a module for the Python programming language[4], which enables to submit jobs to Xgrid controllers and manage them from Python programs. Python is an object-oriented programming language. Fig.11 shows a program with PyXG. In the program, not only job construction steps (the lines 5 and 6) but also steps for connecting to a controller (the lines 2, 3 and 4) and a step for submission (the line 7) are described.

The main differences between PyXG and our approach are as follows.

1. Programs with PyXG are completely procedural while our batch files are basically declarative. The order of definitions is not significant in our approach. Procedural descriptions are used in our batch files if needed as shown in Fig.10. In PyXG, semantic checks and the ordering of job submission must be procedurally described. In our approach, they are done automatically.
2. Programs with PyXG includes everything related to Xgrid while our batch files consist of job specifications only.

The interested reader is referred to [1] for other Xgrid client programs.

V. Conclusions

We proposed a Ruby-based batch language for the grid computing environment Xgrid, and a client program for the language. They solve the problems

```
 1: filelist = Dir.glob("*.txt")
 2:
 3: filelist.each do |f|
 4:   file f.to_s do |t|
 5:     t.agentPathName = f.to_s
 6:     t.localPathName = f.to_s
 7:     t.isExecutable = false
 8:   end
 9: end
10:
11: taskNames = []
12: filelist.each do |f|
13:   taskName = "bc" + f.to_s
14:   taskNames = taskNames | [taskName]
15:   task taskName do |t|
16:     t.command = "/usr/bin/bc"
17:     t.arguments = ["-q", f.to_s]
18:     t.refersTo = [f.to_s]
19:   end
20: end
21:
22: job "job1" do |t|
23:   t.tasks = taskNames
24: end
```

Fig. 10. An RuBLX batch file with dynamically defined tasks

```
1: from xg import *
2: conn = Connection(
           hostname='xgridcontroller',
           password='pass')
3: cont = Controller(conn)
4: g = cont.grid(0)
5: js = JobSpecification()
6: js.addTask('/usr/bin/bc',
              args='bc_script.txt')
7: j = g.batch(js)
```

Fig. 11. A Python program with PyXG

about Xgrid: XML tags as obstacles for human to read and write batch files, consistency management of batch files by users, job management by job ID numbers, and so on. Users with some programming skill can describe batch files using templates. Our approach makes it easy to use Xgrid.

Acknowledgements

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 18700035 from 2006 to 2007.

References

[1] Baden Hughes. Building computational grids with apple's xgrid middleware. In Rajkumar Buyya, Tianchi Ma, Reihaneh Safavi-Naini, Chris Steketee, and Willy Susilo, editors, *ACSW Frontiers*, volume 54 of *CRPIT*, pages 47–54. Australian Computer Society, 2006.

[2] Brian Marick. *Everyday Scripting with Ruby.* Pragmatic Bookshelf, 2007.

[3] David Wood. *Programming Internet Email*. Oreilly & Associates Inc, 1999.
[4] Mark Lutz and David Ascher. *Learning Python*. Oreilly & Associates Inc, 2003.
[5] Tetsuya Suzuki and Takehiro Tokuda. A system for landscape photograph localization. In *ISDA (1)*, pages 1080–1085. IEEE Computer Society, 2006.

Appendix

We explain the detail of our batch language. A batch file includes one or more job definitions, one or more task definitions and zero or more file definitions. The order of definitions is not significant. It can also include any Ruby code anywhere.

Fig.12, Fig.13, Fig.14 and Fig.15 show the syntax of a file definition, a task definition and job definitions respectively. In the figures, non terminal symbols are enclosed with '<' and '>'. A pair of parenthesises followed by a question mark '(X)?' means X is optional. A pair of parenthesises with a vertical bar between them '$(X \mid Y)$' means X or Y.

Each definition has a do-end block with a parameter <*PARAM*>, and the detail of each definition is given there. The order of description in such a do-end block is not significant. A same identifier must be used for <*PARAM*> in a same do-end block.

In the following, we explain the syntax of a file definition, a task definition and a job definition in this order.

Fig.12 shows the syntax of a file definition. A file definition starts with a keyword 'file'. It takes two arguments: a string for a logical file name(<*LOGICAL_FILE_NAME*>) and a block with a parameter. The block specifies the detail of the file. In the following, we use 't' as the parameter. In the block, the value of 't.agentPathName' specifies a path of the file on an agent machine. The content of the file is specified by 't.localPathName' or 't.contents'. The value of 't.localPathName' <*PATH_ON_LOCAL*> specifies the contents by the path of a local file. The value of 't.contents' <*STRING*> specifies the contents by a string. Both of them can not be specified at once. The value of 't.isExecutable' <*EXECUTABLE*> specifies whether the file is executable or not: 'true' for executable, 'false' for not-executable.

Fig.13 shows the syntax of a task definition. A task definition starts with a keyword 'task'. It takes two arguments: a string for a logical task name(<*LOGICAL_TASK_NAME*>) and a block with a parameter. The block specifies the detail of the task. In the following, we use 't' as the parameter. In the block, the value of 't.command' <*PATH_OF_COMMAND*> specifies a path of a command which will run on an agent machine. The value of 't.arguments' <*COMMAND_ARGUMENT_LIST*> specifies command line arguments by an array. The value of 't.environment' <*ENVIRONMENT_HASH*> specifies environment variables and their values by a hash. The keys of the hash are names of environment variables, and their values are values of the environment variables. The value of 't.inputStream' <*LOGICAL_FILE_NAME*> specifies a logical file name whose contents are used as the standard input. The value of 't.dependsOn' <*LOGICAL_TASK_NAME_LIST*> specifies logical task names which the task depends on by an array. The value of 't.refersTo' <*LOGICAL_FILE_NAME_LIST*> specifies logical file names by an array, each of which the task will read. The value of 't.inputFileMap' <*INPUT_FILE_MAP_HASH*> specifies the correspondence between file paths on agents and the contents for this task only by a hash. The keys of the hash are file paths and their values are logical file names.

Fig.14 and Fig.15, which are for previously submitted jobs and jobs to be submitted respectively, show the syntax of a job definition.

In both cases, a job definition starts with a keyword 'job'. It takes two arguments: a string for a logical job name(<*LOGICAL_JOB_NAME*>) and a block with a parameter. The block specifies the detail of a job. In the following, we use 't' as the parameter.

The detail of previously submitted jobs are defined in the block of Fig.14 as follows. The value of 't.id' specifies a previously submitted job ID. The previously submitted job ID is given either by an integer or by a pair of a logical job name and a map file. The pair is specified by 'jobId(<*LOGICAL_JOB_NAME*>, <*MAP_FILE_PATH*>)' where <*LOGICAL_JOB_NAME*> is a logical job name and <*MAP_FILE_PATH*> is the path of a map file which includes the logical job name.

The detail of jobs to be submitted are defined in the block of Fig.15 as follows. The value of 't.mail' <*MAIL_ADDRESS*> specifies an e-mail address to which an e-mail is sent when the job status is changed. The value of 't.taskMustStartSimultaneously' <*TASK_MUST_START_SIMULTANEOUSLY*> specifies whether tasks must start simultaneously or not by a boolean value: 'true' is for yes, and 'false' is for no. The value of 't.minimumTaskCount' <*MINIMUM_TASK_COUNT*> specifies the minimum number of tasks which are needed to start at the same time. The value of 't.dependsOnJobs' <*LOGICAL_JOB_NAME_LIST*> specifies logical job names which the job depends on by an array. The value of 't.files' <*LOGICAL_FILE_NAME_LIST*> specifies logical file names used in the job by an array. The value of 't.tasks' <*LOGICAL_TASK_NAME_LIST*> specifies logical task names in the job by an array.

```
file <LOGICAL_FILE_NAME> do | <PARAM> |
     <PARAM> .agentPathName = <PATH_ON_AGENT>
     ( <PARAM> .localPathName = <PATH_ON_LOCAL> | <PARAM> .contents = <STRING> )
     ( <PARAM> .isExecutable = <EXECUTABLE> ) ?
end
```

Fig. 12. The syntax of a file definition

```
task <LOGICAL_TASK_NAME> do | <PARAM> |
     <PARAM> .command = <PATH_OF_COMMAND>
     ( <PARAM> .arguments = <COMMAND_ARGUMENT_LIST> ) ?
     ( <PARAM> .environment = <ENVIRONMENT_HASH> ) ?
     ( <PARAM> .inputStream = <LOGICAL_FILE_NAME> ) ?
     ( <PARAM> .dependsOn = <LOGICAL_TASK_NAME_LIST> ) ?
     ( <PARAM> .refersTo = <LOGICAL_FILE_NAME_LIST> ) ?
     ( <PARAM> .inputFileMap = <INPUT_FILE_MAP_HASH> ) ?
end
```

Fig. 13. The syntax of a task definition

```
job <LOGICAL_JOB_NAME> do | <PARAM> |
    <PARAM> .id = ( <PREVIOUSLY_SUBMITTED_JOB_ID> |
                              jobId( <LOGICAL_JOB_NAME> , <MAP_FILE_PATH> ) )
end
```

Fig. 14. The syntax of a job definition (case 1)

```
job <LOGICAL_JOB_NAME> do | <PARAM> |
    ( <PARAM> .mail = <MAIL_ADDRESS> ) ?
    ( <PARAM> .taskMustStartSimultaneously = <TASK_MUST_START_SIMULTANEOUSLY> ) ?
    ( <PARAM> .minimumTaskCount = <MINIMUM_TASK_COUNT> ) ?
    ( <PARAM> .dependsOnJobs = <LOGICAL_JOB_NAME_LIST> ) ?
    ( <PARAM> .files = <LOGICAL_FILE_NAME_LIST> ) ?
    <PARAM> .tasks = <LOGICAL_TASK_NAME_LIST>
end
```

Fig. 15. The syntax of a job definition (case 2)