On Using the ICKEPS Simulator Server – A Brief Manual for the Competitors

Stefan Edelkamp
Computer Science Department
University of Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Mark Kellershoff
Computer Science Department
University of Dortmund
mark.kellershoff@uni-dortmund.de

May 2, 2007

Abstract

In this paper we describe the functionality of the ICKEPS Simulation Server, the interface for competitors in the second International Competition on Knowledge Engineering for Planning & Scheduling.

1 Connection Data

Server: ausonia.cs.uni-dortmund.de
Ports: 11001 - 11009

2 Introduction

The ICKEPS Simulator Server is a program using sockets via TCP/IP. Given an existing server application for running algorithms remotely, there was no need to develop a new server. Instead we extended the Vega-Server to run simple file-input-based Linux/Unix applications. Therefore, the simulation-server follows a specific protocol to handle files through TCP. Because of having an interface that is as simple as possible files will be accepted as textinput with a specific format.

IMPORTANT: Throughout the whole document ⟨ and ⟩ just represent variable content! These symbols are only used for readability and are **not** syntactical features.

3 The VegaServer

The ICKEPS Simulation Server has some very simple commands. A sequence of commands and inputs shall produce the desired output.

To demonstrate the usage of this interface or protocol, there are multiple examples. Since everyone can create his own way of opening a socket and communicating with the server, telnet will be the simpliest way to show the server's functionality.

The ICKEPS Simulation Server accepts the following commands:

Shortcut	command	Description
`id`	`identify`	Print Server Identification
`l`	`list`	List all available Simulators
`s`	`select`	Select a simulator
	`info`	Get infos about the selected simulator
`i`	`input`	Upload input to the server
`r`	`run`	Run a Simulation (Be sure to provide the nessessary input beforehand)
`t`	`timeout` ⟨n⟩	Set algorithm timeout ⟨sec⟩
`h`	`help`	Overview of commands
`q`	`quit`	Quit the connection

Some of these commands are not really necessary for competitors. The timeout is applicable only works for algorithms written especially for the server unlike the simple executables. The ICKEPS Simulation Server calls the selected simulator executable. It first redirects the necessary input to it, and then redirects its output to the client, who has called it. To send input to the server it has to be sent as text with a special formatting. The filename itself is not needed. Every simulation run needs the necessary input to be uploaded beforehand.

The `input`-mode is always a sequence like:

- start input-mode with `i`
- upload a parameter (i.e. `plan='<content>'`)

- upload an other parameter (i.e. `options='<experiment name>`)
- ...
- end the input mode with `EOT`

The ICKEPS Simulation Server is only capable of recognizing input in the manner specified. The formating is very simple and always like:
`<parametername>='<parameter content>'`.

The only parameternames possible are:

parameter name	generic desription
plan	A plan or schedule
problem	A problem
domain	A domain where the problem occurs
goals	A file with goals
options	**The name of an experiment**

Each simulator expects a specific subset of these parameters. Refer to the later sections which parameters are needed for a specific simulator. Parameters which were uploaded, but not needed, are simple ignored.

Here is a brief example of a run with the groove simulator. Let's pretend you want to send a plan you have generated for the graphtransformation experiment `append`, then you would have to do the following:

1. Open a connection to the competition-server
2. send: `s GraphTransformation` - the response should be `OK`
3. send: `i` - the response should be `OK`
4. send: `plan='<content of the plan-file>'`
5. send: `options='append'` - The 'append' experiment
6. send: `EOT` - End of Transaction
7. send: `r` - Start a run
8. ⟨Now the server redirects the simulators output to the client.⟩
9. ⟨When the executable is finished the connection is terminated.⟩

To upload all nessessary input files for a simulator, just one input session is needed.

For the cybersecurity-simulator for example you will need three files to execute. The domain file, which contains the cybersecurity domain, a problem file, which contains a problem to investigate and a plan file for the problem. Since the domain-file of cybersecurity will never change, the server uses its local copy and no one has to send it. The input to the server will therefore look like the following code fragment (assuming that you have selected CyberSecurity first and switched the ICKEPS Simulation Server to input-mode):

```
problem='<content of the problem file here>'
plan='<content of the plan file here>'
```

After uploading everything you quit the input-mode with `EOT`. You are now ready to start a cybersecurity-run with `r`.

IMPORTANT: You have to use highquotes (') to mark the beginning and the end of a parameter. **You can use highquotes elsewhere!**

On serverside the three parameters, respective their content, are written to files. The paths to the files are used for the call to the executable. For cybersecurity this will result in executing:

```
 java -jar bams-sim.jar
domainfile problemfile planfile resultfile
```

Everything printed out while a simulator is running will be forwarded to the client. In case a simulator produces a file with output, this file is opened and sent back to the client.

IMPORTANT: Opening a result file and sending it back via TCP/IP sometimes takes a while!

The format for created output files when sending back is:

```
FILE <name>:
<filecontent line1>
<filecontent line2>
< .... >
<filecontent lastline>
ENDFILE
```

CyberSecurity for example always produces one file containing the outcome of the experiment.

```
FILE cyberout:
Success!
ENDFILE
```

4 The Different Simulators

There are six different simulator executables available. The server maps names to simulators like the following table:

⟨Name of the simulator used by the server⟩	⟨Related program⟩
GraphTransformation	Groove
Telescope	Spike
CyberSecurity	Bams
Manufacturing	Manufacturing
Validator	Validate
PowerSupply	PSR

Each of them is selectable through either `select <name>` or `s <name>`.

The rest of this section will explain which parameter is used for which file content regarding a specific simulator.

IMPORTANT: The ICKEPS Simulation Server just redirects the input to files and the files' paths to the simulator executable. It does not change anything in the inputstream.

IMPORTANT: The two simulators Telescope and GraphTransformation use the `options='...'` parameter to choose the experiment. (The other simulators get the experiment through uploads.)

The subsection 'Execution' will explain how the executables are called by the server. That will make it easy to reproduce the results 'at home'. (Unfortunately, some simulators are not freely available, so we will just explain, how they are used through the server.)

4.1 GraphTransformation with Groove

Groove is a tool to exlore a graph entirely by applying all possible sequences of rules.

4.1.1 Input

This simulator accepts the `plan` parameter and the `options` parameter. The plan parameter contains a sequence of rules and the options parameter contains the name of the experiment. (For example: 'append') example:

```
plan='next
next
next
append
return'
```

4.1.2 Output

Groove generates all possible ending states as gxl-files. These files are sent seperately to the client. Each of them begin with `FILE GROOVE s8:`, where `s8` is the coresponding statenumber generated by groove, and end with `ENDFILE`.

4.1.3 Execution

```
 java -classpath$SIMPATH/groove-edelkamp.jar:
$SIMPATH/lib/castor-0_9_5_2.jar:
$SIMPATH/lib/jgraph-5_2_1.jar:
$SIMPATH/lib/xerces-2_6_0-xercesImpl.jar:
$SIMPATH/lib/jgraphaddons-1_0_4.jar:
$SIMPATH/lib/junit-4.2.jar
groove.Generator -v 0 -x controlled:<planfile>
-f <outputfileprefix> <experiment>
```

4.2 CyberSecurity with Bams

The Cyber-Security simulator validates if a system can be compromised or is safe.

4.2.1 Input

It takes a problem file and a plan as it's input.

```
problem='<content>'
plan='<content>'
```

4.2.2 Output

The output is either `Success!` or a message of the goals failed.

4.3 Validator

The validator is very similar to the cybersecurity simulator. The cybersecurity simulator checks a plan for a problem in the specific cybersecurity domain while the validator checks a plan in any given domain. The specific test-domain must also be uploaded.

4.3.1 Input

Therefore the input parameters are the same as for cybersecurity.

```
domain='<content>'
problem='<content>'
plan='<content>'
```

4.3.2 Output

The output of validate is the outcome of applying the given plan to the problem.

4.4 Scheduling with Spike

Think of a huge telescope used for viewing special observations. Every observation is viewable only through a specific period of time. Since there are many observations for one night you have to find the best schedule viewing the most important observations.

4.4.1 Input

The Spike simulator takes a schedule as input via the `plan` parameter and an experiments' name through the `options` parameter.

```
plan='<content>' options='<experiment-name>'
```

4.4.2 Output

The output of spike are two files. One file that shows the report on the schedule and the other one showing the gaps that are between observations while running the schedule.

```
FILE SCHEDREP:
<line 1 of the file>
<line 2 of the file>
...
<line n of the file>
ENDFILE
FILE GAPREP:
<line 1 of the file>
<line 2 of the file>
...
<line n of the file>
ENDFILE
```

4.5 Power Supply Restoration

The Power Supply Restoration Simulator checks if a plan for restoring a faulty power line is valid. This domain uses extra constraints for applying a plan.

4.5.1 Input

This simulator takes a network (which includes a set of faulty switches) and a plan to restore power lines.

```
problem='<content of a network file>'
plan='<content of a plan file>'
```

4.5.2 Output

The Psr simulator produces a plan as formatted output where the last step contains the plan validity.

4.6 Manufacturing

The manufacturing simulator is used to prove manufacturing processes.

4.6.1 Input

The input contains a manufacturing domain, a problem and a set of goals to be reached.

```
domain='<content of a domain file>'
problem='<content of a problem file>'
goals='<content of a goals file>'
```

4.6.2 Output

The output is generated by the executable while it is running. It consists of the plan evaluation and information on the goals missed or reached.

4.6.3 Execution

```
simulator.linux -D <domain.file>
-P <problem.file>
-G <goal.file>
-E oplan_pro.linux
```

5 Clients

There are more ways to access the server. One way as explained is telnet and an other one is using your own automated client. To show shortly how this could be developed, examine the following Java-Client as example. The source (and the created binary with a start script) is available on the internet page of ICKEPS.

5.1 A Java-Client

The Java-Client for accessing the ICKEPS Simulation Server has a simple syntax. It consists of one class named VegaSimTester. It can be startet with the following command.

```
VegaSimTester <SimulatorShortcut>
     <path-to-file-containing-all-inputs>
     <URL> <port>
```

For example:

```
VegaSimTester G /path/to/inputfile
            ausonia.cs.uni-dortmund.de
            11003
```

The Java-Client will select Graph Transformation and upload everything contained in inputfile to our server and start a run. Everything the server displays will be displayed in the console.

Some methods in this class are rather unimportant for communicating with the server like a `readFile()` method or a Constructor. These methods should explain themselves.

There are two methods interesting for the competition.

The first one is `createSocket()` which opens a socket to communicate with the server. This method simply creates a socket and binds the streams.

```
public void createSocket() throws IOException{

 mySocket = new Socket(hostName, port);
 bin = new BufferedReader(
        new InputStreamReader(
          mySocket.getInputStream()));
 bout = new BufferedWriter(
         new OutputStreamWriter(
          mySocket.getOutputStream()));
}
```

The other method is `communicate()` which manages everything necessary to use the server. It looks a bit like the communication via telnet does but in an automated way. The methods `send(String)`, `receive()` and `waitForResponse()` really do what they look like. The send() method just sends a string to the server, `receive()` just receives a response from the server and `waitForResponse()` just waits for any input to be available.

```
public void communicate() throws IOException{

 String answer="";

 //select
 send("s "+algoName+"\n");
 waitForResponse();
 answer=receive();
```

```
 if (!answer.equals("OK"))return;

 //input
 send ("i\n");
 waitForResponse();
 answer=receive();
 if (!answer.equals("OK"))return;

 //sendInput
 String[] toSend=input.split("\n");
 for (String r:toSend){
  send(r+"\n");
 }
 send("EOT\n");
 waitForResponse();
 answer=receive();
 if (!answer.equals("OK"))return;

 //run
 send("r\n");
 boolean running=true;
 long time=System.currentTimeMillis();
 long timeout=time+offset;
 while(running){
  time=System.currentTimeMillis();
  if(bin.ready()){
   answer=receive();
   timeout=time+offset;
  }
  if (time>=timeout){
   running=false;
  }
 }
}
```

This method simply selects an algorithm via the `s ...` command, switches the server to input-mode via the `i` command and uploads a file line-by-line. The upload-file content should be formatted like the input with telnet. For example:

```
plan='<original file content>'
options='<experiment-name>'
```

It ends input-mode via `EOT` and starts the run via the `r` command. This procedure is the same for all different simulators.

6 Troubleshooting

If you encounter any problems working with the server or this document please feel free to send an email to us. Any suggestions are welcome.